Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica(v.o.)

Tesi di laurea

Sistemi embedded soft real-time basati su kernel Linux

Relatore: Ch.mo Prof. Mauro Migliardi

Laureando: Enrico Michelon, matricola 421581/IF

Anno accademico 2011/2012

Indice

Indice **i**

1 Introduzione **1**

2 Sistemi embedded real-time **3**
2.1 Sommario . . . 3
2.2 Definizione . . . 3
2.3 Job e Task . . . 4
2.3.1 Parametri temporali . . . 4
2.3.2 Parametri prestazionali . . . 5
2.4 La schedulazione . . . 7
2.4.1 Problematiche . . . 10
2.4.2 Algoritmi di schedulazione . . . 12
2.5 Sistemi embedded . . . 15

3 Il soft real-time **19**
3.1 Sommario . . . 19
3.2 Problematiche legate al soft real-time . . . 19
3.3 Primi approcci al soft real-time . . . 21
3.3.1 Sistemi general purpose . . . 21
3.3.2 Sistemi hard real-time . . . 22
3.4 Peculiarità di un sistema soft real-time . . . 23
3.5 Overload . . . 24
3.5.1 Controllo di ammissibilità . . . 25
3.5.2 Degrado prestazionale . . . 26
3.6 Overrun e protezione temporale . . . 28
3.6.1 Fair scheduling . . . 29
3.6.2 Resource reservation . . . 30
3.7 Il multi-thread . . . 33
3.7.1 Algoritmo di Deng e Liu . . . 34
3.7.2 BSS . . . 36
3.7.3 Partizionamento delle risorse . . . 38
3.8 Sincronizzazione . . . 38

3.8.1 BWI . . . 40
3.9 Recupero delle risorse . . . 42
3.9.1 CASH e GRUB . . . 42
3.9.2 Algoritmi semplificati di recupero . . . 44
3.10 Qualità del servizio . . . 47
3.10.1 Il modello Q-RAM . . . 47
3.10.2 QoS e gestione dinamica delle risorse . . . 48
3.10.3 Smooth rate adaptation . . . 49
3.11 Scheduling con feedback(adattabilità) . . . 50
3.11.1 FC-EDF . . . 50
3.11.2 Adaptive reservation . . . 51
3.11.3 Adattamento a livello di applicazione . . . 52
3.11.4 Stimatori di carico . . . 53

4 Il kernel Linux **55**
4.1 Sommario . . . 55
4.2 Storia di Linux . . . 55
4.3 Concetti base . . . 56
4.3.1 Tipologie di kernel . . . 57
4.3.2 I contesti di esecuzione . . . 58
4.3.3 Kernel e applicazioni utente . . . 59
4.3.4 Linux e il soft real-time . . . 61
4.4 Processi e thread . . . 62
4.4.1 Implementazione . . . 63
4.4.2 Manipolazione di processi e thread . . . 64
4.5 Schedulatore . . . 66
4.5.1 Parametri di schedulazione . . . 68
4.5.2 Parte principale dello schedulatore . . . 69
4.5.3 Algoritmi di schedulazione in Linux . . . 71
4.6 Misura del tempo . . . 74
4.7 Interruzioni . . . 77
4.7.1 Gestori di interruzioni . . . 78
4.7.2 Top half . . . 78
4.7.3 Bottom half . . . 80
4.8 Sincronizzazione . . . 83
4.8.1 Operazioni atomiche . . . 84
4.8.2 Spin locks . . . 85
4.8.3 Semafori . . . 85
4.8.4 Futex . . . 86
4.8.5 Sequential locks . . . 87
4.8.6 Read-copy update . . . 87
4.9 Gestione della memoria . . . 89
4.9.1 Suddivisione della memoria . . . 89
4.9.2 Manipolazione della memoria . . . 90

4.9.3 SLAB, SLOB e SLUB . . . 91
4.9.4 Altri metodi di allocazione . . . 92
4.9.5 Gestione della memoria in spazio utente . . . 93
4.10 I/O scheduler e caching del disco . . . 94
4.10.1 Il virtual filesystem . . . 94
4.10.2 Scheduler I/O per dispositivi a blocchi . . . 95
4.10.3 Il page cache . . . 98

5 Android **101**
5.1 Sommario . . . 101
5.2 Storia . . . 101
5.3 Struttura generale . . . 102
5.4 Rapporto con il soft real-time . . . 104
5.4.1 Dalvik VM . . . 104
5.4.2 Zygote . . . 107
5.4.3 Garbage collector . . . 108
5.4.4 Componenti delle applicazioni . . . 110
5.4.5 Task e back stack . . . 114
5.4.6 Processi . . . 115
5.4.7 Thread . . . 118
5.4.8 Wakelock . . . 120
5.4.9 Programma di compatibilità . . . 121
5.4.10 Utilizzo di codice nativo . . . 123
5.5 Prestazioni real-time . . . 128

6 Ubuntu phone **131**
6.1 Sommario . . . 131
6.2 Storia . . . 131
6.3 Struttura . . . 132
6.3.1 Dispositivi target . . . 133
6.3.2 Kernel . . . 133
6.3.3 Esecuzione dei programmi . . . 134
6.3.4 Librerie Qt . . . 134
6.3.5 Interfaccia utente . . . 137
6.4 Prime valutazioni . . . 140

7 Conclusioni **141**
7.1 Sviluppi futuri . . . 142

Elenco delle tabelle **143**

Elenco delle figure **145**

Bibliografia **147**

Capitolo 1

Introduzione

L'elaborazione real-time (RT) [1] ha acquisito un'importanza cruciale all'interno della nostra società dal momento in cui un numero sempre crescente di sistemi complessi ha iniziato ad essere gestito, completamente o in parte, tramite l'ausilio di calcolatori.

Alcuni esempi di applicazioni reali vedono il controllo di impianti nucleari o del traffico aereo, il monitoraggio di sottosistemi avionici o automotive o la gestione di robot o sistemi in ambito militare.

In tempi più recenti [2] i sistemi RT hanno ampliato il loro raggio d'azione, estendendo la loro presenza in nuovi settori quali gli apparati di riproduzione multimediale, la realtà virtuale e giochi elettronici interattivi. A differenza dei precedenti, che comprendevano solamente ambienti critici sotto il profilo della sicurezza, questi ambiti applicativi, indicati con il nome di sistemi *soft real-time*, sono caratterizzati da un comportamento fortemente dinamico e una maggiore flessibilità per quanto concerne il rispetto dei vincoli temporali, tale per cui la violazione di uno di questi vincoli non determina generalmente situazioni di alto rischio per l'incolumità di cose o persone ma solamente un degrado prestazionale del sistema stesso.

Il kernel Linux, malgrado sia stato concepito inizialmente per sistemi general-purpose, ha conosciuto negli ultimi anni una rapida diffusione(in modo particolare grazie alla crescita esponenziale dei sistemi Android) [55] come elemento cardine di vari sistemi operativi su dispositivi embedded in ambiente soft real-time.

Obiettivo della tesi è quindi quello di indagare quali caratteristiche intrinseche del kernel Linux ne abbiano reso possibile l'impiego in questo particolare ambito e come queste siano state impiegate in alcuni sistemi operativi reali, evidenziando le principali differenze tra gli approcci adottati nei vari casi.

La presente tesi sarà quindi sviluppata secondo lo schema seguente:

- Nel capitolo 2 verrà offerta una panoramica sulle caratteristiche e le problematiche relative ai sistemi real-time embedded.

- Nel capitolo 3 saranno approfondite le peculiarità inerenti al soft real-time, evidenziandone le criticità principali e alcune possibili soluzioni offerte in campo teorico.
- Nel capitolo 4 analizzeremo la struttura del kernel Linux, soffermandoci principalmente su quegli aspetti che ne influenzano, in positivo o negativo, l'adattabilità ad ambienti di tipo soft real-time.
- Il capitolo 5 contiene una trattazione del sistema operativo Android focalizzata a determinarne le componenti, ereditate dal kernel Linux o integrate ex-novo, mirate a garantire un supporto quanto più completo possibile ai dispositivi embedded con requisiti soft real-time.
- Il capitolo 6 sarà invece dedicato all'esplorazione delle caratteristiche del nuovo sistema operativo Ubuntu phone, cercando per quanto possibile di evidenziarne le differenze strutturali rispetto ad Android e valutandone qualitativamente l'impatto, positivo o meno, sulle prestazioni real-time del sistema.
- Nel capitolo 7 verranno riportate le conclusioni dell'analisi svolta e i possibili sviluppi futuri.

Capitolo 2

Sistemi embedded real-time

2.1 Sommario

Verrà offerta una definizione di real-time e delle componenti basilari con cui le sue caratteristiche vengono modellate a livello teorico. Successivamente, si espongono alcune problematiche connesse alla gestione dei sistemi real-time. Infine, si accenna brevemente agli aspetti principali legati ai dispositivi embedded.

2.2 Definizione

In prima istanza possiamo rilevare che, nel senso comune, un sistema real-time è associato ad un dispositivo capace di reagire a determinati eventi entro un intervallo di tempo limitato.

Immaginiamo un computer dedicato al controllo di un robot: possiamo facilmente intuire che il tempo necessario alla cpu per reagire ad un evento proveniente dall'ambiente in cui il robot stesso opera diviene un parametro cardine per stabilire le capacità e le prestazioni dell'intero apparato. Se ad esempio il robot trova un ostacolo imprevisto sul suo percorso, l'azione di bloccaggio o di modifica della traiettoria deve essere eseguita entro un intervallo di tempo massimo, dipendente sia dalla distanza dall'ostacolo che dalla velocità del robot stesso, o altrimenti una collisione sarà inevitabile, con conseguenze che, a seconda dell'ambito d'impiego, possono risultare più o menio gravi.

Un sistema in tempo reale può essere quindi definito come un apparato elaborativo in cui le varie attività computazionali devono essere realizzate entro predefiniti vincoli temporali e il cui grado di funzionalità non dipende soltanto dalla correttezza formale dei risultati delle elaborazioni ma anche dal tempo necessario a produrli.

Inoltre, il termine *real* richiama il fatto che il tempo di sistema deve essere adattabile alla scala temporale utilizzata per misurare gli intervalli di tempo

tipici dell'ambiente misurato; ad esempio, se si deve gestire un fenomeno nella scala dei microsecondi, anche il sistema dovrà presentare caratteristiche di risoluzione, precisione temporale e tempi di risposta almeno dello stesso ordine di grandezza, se non addirittura superiori.

2.3 Job e Task

I modelli utilizzati per i sistemi real-time sono basati sui concetti di job e task [3]; con il primo ci si riferisce alla singola unità elaborativa che può essere schedulata ed eseguita da un processore, mentre il task (o processo) risulta essere pari ad un insieme, finito o infinito, di job che concorrono alla realizzazione di una data funzione.

2.3.1 Parametri temporali

Il comportamento dei job viene generalmente espresso tramite alcuni parametri temporali, illustrati in Figura 2.1 e brevemente presentati nell'elenco seguente:

- **Release time**: istante nel quale il job è pronto per essere eseguito, ed è quindi in attesa di vedersi assegnato dal sistema una frazione della capacità elaborativa dei processori presenti.
- **Start time**: momento in cui ha inizio l'effettiva esecuzione del job da parte dell'elaboratore
- **Computation time**: corrisponde all'intervallo necessario a completare l'intero job, nelle ipotesi preliminare che il job stesso disponga di tutte le risorse necessarie al suo completamento e la sua esecuzione non venga interrotta in alcun modo.
- **Finishing time**: istante in cui termina l'esecuzione del job.
- **Absolute deadline**: istante di tempo espresso in coordinate assolute entro il quale il job deve essere completato.
- **Relative deadline**: simile alla precedente ma espressa come intervallo relativo misurato a partire dall'istante di rilascio.

Tutti questi parametri vengono poi usati per definire le specifiche del sistema real-time; queste ultime [20] sono generalmente espresse non con valori esatti ma con intervalli di valori possibili, in quanto si deve tenere conto di tutte le cause di errore che possono influenzarne la misurazione.

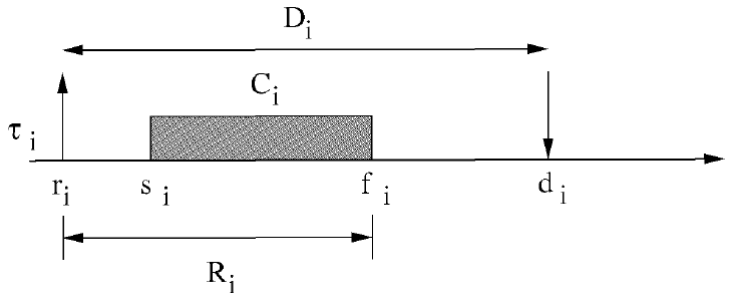


Figura 2.1: Parametri temporali dei job

2.3.2 Parametri prestazionali

Al precedente gruppo viene sovente affiancata una seconda partizione di parametri temporali da utilizzare come indici della qualità delle prestazioni real-time del sistema stesso; di questi gli ultimi tre [20] sono pensati in modo particolare per la valutazione dei sistemi soft real-time:

- **Response time**: equivale all'intervallo compreso tra release time e finishing time. Può fornire indicazioni parziali sulla dimensione dei tempi di risposta ad eventi esterni.
- **Slack time o laxity**: è la differenza tra la deadline assoluta e l'istante di terminazione. Il suo valore indica di quanto l'esecuzione di un job possa essere ulteriormente ritardata nel tempo pur rispettando il vincolo imposto dalla deadline associata. Un eventuale valore negativo del parametro indica ovviamente la violazione dei vincoli temporali legati al task stesso.
- **Lateness**: è esattamente l'opposto della slack time e viene impiegato per stabilire con quanto ritardo rispetto alla propria deadline un job venga completato.
- **Tardiness**: è pari al massimo valore tra zero e la lateness e si utilizza esclusivamente per individuare i job che violano la deadline loro associata.
- **Miss rate**: è la percentuale di job (soft) che terminano la loro esecuzione oltre la deadline abbinata.
- **Loss rate**: indica il rapporto tra i job (soft) ignorati o solo parzialmente eseguiti e il numero totale di job del sistema.
- **Invalid rate**: è pari alla somma dei due parametri precedenti e offre quindi un'indicazione della percentuale di job che o non producono un risultato o ne forniscono uno di utilità solo parziale.

Per quanto riguarda la descrizione del comportamento dei task, risultano importanti i due seguenti criteri:

- Si distinguono i task a seconda della periodicità legata ai job che li compongono, individuando quindi tre diverse categorie:
  - **periodici**: l'intervallo tra i release time di due job consecutivi appartenenti allo stesso task è sempre pari ad un valore costante T detto periodo del task. Tale caratteristica permette di stabilire relazioni deterministiche tra alcuni parametri temporali abbinati ai job: indicato con J_i l'i-esimo job e detta Φ (fase) il release time di J_1, gli istanti di rilascio dei successivi job si ricavano semplicemente da $\Phi + (i-1) * T$. In questo caso si è soliti porre la deadline relativa pari al periodo T.
  - **aperiodici**: presi due job consecutivi tali che j_1 preceda temporalmente j_2, la stessa relazione deve sussistere tra i rispettivi release time.
  - **sporadici**: rispetto ai precedenti, deve inoltre esitere un intervallo minimo non nullo tra release time di job consecutivi, detto tempo minimo di inter-arrivo.
- I task vengono partizionati in quattro gruppi in base ai vincoli temporali dei propri job:
  - **Hard**: tutti i job che compongono il task devono essere obbligatoriamente completati entro le rispettive deadline; in caso contrario, l'intero sistema verrebbe a trovarsi in un stato di malfunzionamento tale da renderlo totalmente inutilizzabile o comportare gravi rischi per gli utenti che ne fanno uso.
  - **Firm**: All'interno del task, solo ad un determinato sottogruppo di job è permesso violare le deadline associate.
  - **Soft**: In questo gruppo rientrano i task la cui qualità dei risultati decresce gradualmente all'aumentare del tempo di risposta. Si distinguono due scenari possibili: se i job non sono abbinati ad alcuna deadline, il sistema potrà incrementare le proprie prestazioni mirando a diminuire i tempi di risposta associati a ciascuno di essi; viceversa, i job sono associata ad una *soft deadline*, così chiamata poichè il rispetto della stessa comporta il raggiungimento delle massime prestazioni per il sistema, mentre in caso contrario si verifica un degrado delle stesse, quantificabile grazie ad una predefinità funzione di utilità.
  - **Non real-time**: Questo ultimo gruppo comprende i task in cui la correttezza dei risultati forniti dai job è totalmente indipendente dalla quantità di tempo impiegata per ottenerli.

In realtà quest'ultima suddivisione viene basata in letteratura anche su altri criteri distintivi [3] [8], pur manenendo sostanzialmente invariate le categorie individuate.

In alcuni contesti risulta inoltre utile distinguere [4] tra processi *I/O bound* e *cpu-bound*.

I primi spendono la maggior parte del tempo nell'effettuare richieste di I/O e attenderne i dati in risposta. Il loro sviluppo è quindi caratterizzato da brevi intervalli di esecuzione inframmezzati da lunghi periodi di attesa: esempi possono essere tutte le applicazioni legate all'interfaccia utente, che attendono input da quest'ultimo.

I secondi, invece, spendono la maggior parte del tempo nell'esecuzione del codice quindi proseguono fino a che non subiscono prelazione. Generalmente sono attivati meno frequentemente ma per periodi più lunghi rispetto ai task appartenenti all'altro gruppo.

I processi sono caratterizzati generalmente da comportamenti intermedi rispetto a questi due estremi. Il sistema deve quindi riuscire a soddisfare due requisiti in conflitto tra loro: garantire risposte rapide ai primi (ossia mantenere una latenza bassa) e massimizzare l'utilizzazione del sistema nel caso dei secondi. Lo schedulatore I/O utilizzato attualmente in Linux,(sezione 4.10.2) tende ad esempio a privilegiare i processi di tipo I/O per garantire una migliore esperienza interattiva.

Infine, molte delle caratteristiche proprie dei job possono essere attribuite anche ai task; nel proseguio della trattazione, ad esempio, quando parleremo di interrompibilità di un task ci riferiremo al fatto che tutti i suoi job presentano questa peculiarità.

2.4 La schedulazione

Quando ci riferiamo ad un sistema real-time, il problema principale è quello della schedulazione dei task, ossia stabilire in ogni istante quali processi mandare in esecuzione, quando e per quanto tempo. A renderne particolarmente complessa la trattazione (il problema, nella sua formulazione generale, è NP-completo e quindi non trattabile computazionalmente [1]), oltre ai requisiti temporali precedentemente introdotti, rientrano anche altre due tipologie di vincoli:

- **Vincoli di precedenza**: in alcune applicazioni i task non possono essere eseguiti secondo un ordine arbitrario ma devono sottostare a precise relazioni di precedenza fissate durante la loro progettazione (generalmente espresse mediante un *grafo di precedenza*). Questo comporta il fatto che, in ogni istante, i processi da portare in esecuzione potranno essere scelti solo da un sottoinsieme ristretto di quelli attivi all'interno del sistema.

- **Vincoli di risorse**: ogni processo o job necessita di alcune risorse per essere eseguito, quali la cpu, la memoria, file, strutture dati o registri di dispositivi periferici; mentre alcune di esse possono essere private (ossia dedicate esclusivamente ad un unico processo), nella maggior parte dei casi queste sono condivise fra più processi simultaneamente.

  Al fine di conservare la consistenza dei dati, deve però essere previsto un qualche meccanismo di mutua esclusione che consenta al più ad un solo task di accedere alla risorsa in un dato istante; questo comporta di conseguenza che altri contendenti debbano attendere(*waiting*) che la risorsa torni libera prima di poter proseguire la loro elaborazione: in questo particolare stato, quindi, non possono essere selezionati per essere mandati in esecuzione su una delle cpu presenti nel sistema.

In modo più rigoroso, quindi, possiamo affermare che, una volta specificati un insieme di n task $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, un insieme di m processori $P = \{P_1, P_2, ..., P_m\}$ e un insieme di s tipi di risorse $R = \{R_1, R_2, ..., R_s\}$, il problema della schedulazione consiste nel determinare un algoritmo in grado di assegnare processori e risorse ai task presenti in modo tale da completarli tutti nel rispetto dei vincoli specificati (*schedulazione fattibile* [20]).

Per ridurne la complessità dell'analisi, spesso si operano delle restrizioni, riformulando il problema nell'ipotesi di un singolo processore, di mancanza di vincoli di precedenza o di risorse condivise o contemplando la presenza di una sola tipologia di task (ad esempio considerando solo task periodici)

Inoltre, viene associata ai task e ai job tutta una serie di ulteriori parametri, tra i quali i più importanti sono:

- **Priorità**: è un numero assegnato a ciascun job attraverso il quale viene definito l'ordine con cui questi verranno eseguiti; una priorità elevata corrisponderà quindi ad una maggiore probabilità per il job di vedersi assegnata una cpu per proseguire o dare il via alla propria elaborazione.

- **Importanza**: è un parametro numerico deve dà una misura di quanto sia fondamentale per l'intero sistema mantenere o meno l'esecuzione di un dato processo. A differenza della priorità, che regola l'ordine con cui i job vengono portati in esecuzione, il valore rimanda all'importanza del task in funzione del mantenimento della funzionalità dell'intero sistema.

- **Funzione di utilità**: è una funzione matematica che serve a descrivere il livello di utilità del risultato fornito da un job in funzione dell'istante in cui questo viene prodotto.Benché possa essere stabilita in modo arbitrario, se ne individuano quattro principali tipologie, ciascuna legata ad una delle categorie in cui sono distinti i job. Riferendoci anche alla Figura 2.2 possiamo notare che:

  - se il task viene completato prima della deadline associata, l'importanza rimane costante, pur presentando un valore maggiore per i processi di tipo hard e degradando man mano fino a quelli non real-time
  - una volta superata la deadline, le differenze divengono più marcate: per gli hard real-time l'importanza precipita a meno infinito, segnalando quindi una anomalia grave del sistema, mentre per quelli firm essa va a zero, rendendo quindi nulla l'importanza di un eventuale risultato restituito.

    Nel caso dei task soft real-time essa degrada in modo continuo, simulando quindi un calo prestazionale non improvviso come nei casi precedenti, ma distribuito su un intervallo di tempo finito. Infine i processi non real-time, come già accennato precedentemente, non risentono in alcun modo dell'istante nel quale vengono completati.

- **Interrompibilità(*preemption*)**: un job è detto interrompibile se la sua esecuzione può essere interrotta dallo scheduler nel caso si renda disponibile nel sistema un processo a maggiore priorità. In caso contrario, il job viene sempre eseguito ininterrottamente fino alla sua terminazione, e quindi le decisioni dello schedulatore si situano sempre nell'intervallo temporale compreso tra la conclusione di un job e l'inizio di quello successivo. Questa caratteristica è fortemente interconnessa con il cambio di contesto (*context switch*), ovvero la procedura per il salvataggio delle informazioni che descrivono lo stato del processo interrotto al fine di poterne riprendere correttamente l'esecuzione in un secondo momento, in quanto il tempo impiegato per attuarla può influire più o meno pesantemente sulla valutazione delle schedulazioni reali.

- **Utilizzazione**: per definire il carico di lavoro del processore si utilizza la funzione di *processor demand* $g(t_1, t_2)$, definita nell'intervallo $[t_1, t_2]$ e tale da sommare tutte le elaborazioni richieste all'istante t_1 o successivi che devono essere completate entro l'istante t_2. Per un task essa corrisponde quindi alla somma dei tempi di esecuzione dei job i cui release time e deadline sono compresi nell'intervallo temporale considerato; basterà poi sommare i valori di tutti i task attivi nel sistema per ricavare l'ammontare della capacità elaborativa totale da essi richiesta al sistema.

  Il carico di lavoro vero e proprio viene infine calcolato come rapporto tra la processor demand e l'intervallo di tempo considerato per determinarla.

  Nel caso dei processi periodici tale grandezza viene altrimenti indicata come *utilizzazione* U poichè assume una forma particolarmente sem-

plice se calcolata in riferimento al periodo del task, risultando pari al rapporto tra il tempo di esecuzione massimo relativo ai job che lo compongono e la durata del periodo stesso. Essa rappresenta quindi la larghezza di banda elaborativa massima che il task richiede alle cpu per la sua esecuzione.

La definizione di utilizzazione [2] è stata estesa poi anche a processi non periodici, definendola come il valore U per cui la $g(t_1, t_2)$ risulta sempre minore o uguale a $(t_2 - t_1) * U$ ed esiste un intervallo $[t_a, t_b]$ interno al precedente in cui questa espressione vale con il solo segno di uguaglianza.

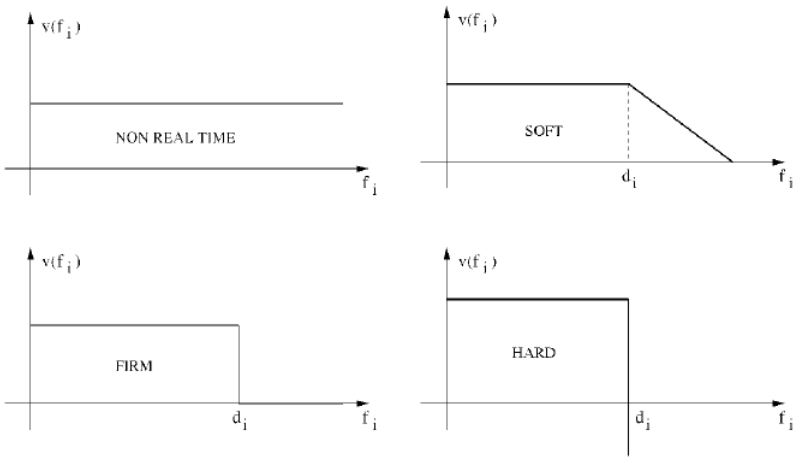


Figura 2.2: Funzioni di utilità associate ai vari tipi di job

2.4.1 Problematiche

I principali fattori di criticità nell'analisi della fattibilità di schedulazione sono legati alla determinazione del tempo di esecuzione dei job.

Quest'ultimo, nei sistemi reali, è definito da un elevato numero di fattori i cui effetti non sono facilmente quantificabili a priori o presentano un'elevata varianza; alcuni di essi sono legati a comportamentii anomali dei task, mentre altri sono determinati o da meccanismi hardware o dalla struttura del kernel del sistema operativo. Queste fonti di indeterminazione sono qui di seguito riportate:

- **overload**: indica una situazione in cui il tempo di elaborazione richiesto dai task è superiore alla capacità elaborativa della cpu presente, ovvero quando $g(t_1, t_2) > t_2 - t_1$, ossia quando il valore di utilizzazione è superiore all'unità. Questa condizione è chiaramente problematica in quanto i vari processi da eseguire tendono ad accumularsi nelle code di

attesa del sistema e a presentare tempi di risposta sempre più elevati. Nel caso di task real-time, quindi, avremo una crescente violazione delle deadline che, nel caso di sistemi soft real-time, si tradurrebbe in un progressivo calo delle prestazioni del sistema e della qualità del servizio fornito.

- **overrun**: mentre l'overload coinvolge l'intero sistema, l'overrun è invece legato al singolo task, e si manifesta quando esiste un intervallo in cui la $g(t1, t2)$ riferita al singolo task supera la larghezza di banda di cui può disporre. Le cause che la generano sono una frequenza di arrivo dei job superiore o tempi di esecuzione più lunghi rispetto a quelli preventivati in sede di progetto.

  Sia l'overrun che l'overload, se non opportunamente gestite, possono dare luogo a situazioni quali il *"Domino effect"*: se infatti si verifica l'attivazione di un task a run-time o comunque una non prevista situazione di overrun che comporta una maggiore utilizzazione del tempo cpu, può avvenire che tutti i restanti processi a minore priorità attivi siano posticipati nella loro esecuzione e vadano a violare sistematicamente le deadline, causando un significativo degrado prestazionale del sistema.

- **DMA**: è una tecnica utilizzata da molte periferiche per trasferire dati da e verso la memoria senza coinvolgere la cpu per il controllo del processo; generalmente però processore e dispositivi di I/O condividono lo stesso bus per comunicare con la memoria e quindi la cpu può rimanere bloccata se il DMA è già impegnato in un trasferimento dati, introducendo così ritardi non prevedibile nell'esecuzione dei task. Per risolvere il problema è possibile disabilitare il DMA o fare in modo che gli accessi alla memoria siano gestiti secondo time-slice riservate alternativamente alla cpu o al DMA, ottenenedo per contro una utilizzazione delle risorse meno efficiente.

- **Cache**: ci si riferisce a meccanismi hardware o software che permettono di mantenere in memoria le porzioni di codice accedute più frequentemente o i dati che si prevede saranno utilizzati in un prossimo futuro, in modo da averli disponibili in tempi più rapidi. I meccanismi di cache consentono usualmente un notevole incremento del throughput del sistema, ma aumentano anche la variabilitàdel tempo con ll quale uno stessa porzione di codice può essere eseguita.

- **Interruzioni**: costituiscono il meccanismo con cui il kernel comunica con le periferiche di sistema. Generalmente ad esse viene attribuita una priorità maggiore rispetto a quella dei processi di sistema (i dispositivi spesso hanno dei vincoli temporali di funzionamento molto stringenti) e quindi la frequenza con cui vengono richiamate aumenta

l'imprevedebilità dei tempi di esecuzione degli stessi processi. La situazione potrebbe essere migliorata introducendo all'interno del kernel una differente politica di gestione delle interruzioni [1].

- **Chiamate di sistema**: dovrebbero essere caratterizzate da un tempo di esecuzione limitato ed essere interrompibili, in modo da non ritardare in modo imprevedibile l'esecuzione di attività critiche.

- **Gestione della memoria**: influisce sulla prevedibilità del tempo di esecuzione dei job attraverso l'indeterminazione causata da meccanismi di paginazione e di swap della memoria. Soluzioni tipiche adottate sono un partizionamento statico della memoria (se ne riserva una porzione per ogni processo attivato) o la disattivazione dello swap, con la contropartita di ottenere un sistema meno flessibile e un sottoutilizzo della memoria di sistema.

- **Sincronizzazione**: si è visto in precedenza che l'accesso alle risorse condivise deve essere regolato da un qualche meccanismo di mutua esclusione; la sua introduzione, se non opportunamente gestita, può dare luogo a due fenomeni capaci di incrementare in modo indefinito i tempi di esecuzione di un job:

  - **Deadlock**: si attua quando due o più job detengono ciascuno in mutua esclusione una risorsa necessaria all'avanzamento dell'esecuzione degli altri job e contemporaneamente necessitano di una delle risorse attualmente detenute da altri. Si instaurano così situazioni di stallo la cui gestione vedremo in modo più dettagliato nel prossimo capitolo.
  - **Inversione di priorità**: si verifica ad esempio quando un job A a bassa priorità acquisisce per primo una risorsa R condivisa in mutua esclusione con un secondo job B, che per accedervi è forzato ad attenderne il rilascio per un tempo pari almeno alla lunghezza della sezione critica (blocco di codice compreso tra le operazioni di acquisizione e rilascio della risorsa) del primo job, anche nel caso sia caratterizzato da una maggiore priorità.

    La situazione diviene anche più critica in presenza di altri job $C_1, C_2, ..., C_n$ di priorità intermedia ai precedenti; questi andrebbero a prelazionare A, posticipandone ulteriormente il completamento e comportando per B un ritardo impredicibile e tale da non poter essere limitato superiormente.

2.4.2 Algoritmi di schedulazione

La varietà degli algoritmi proposti per la schedulazione di processi real-time è vastissima.

Se ne possono riconoscere varie tipologie, così caratterizzate:

- **Preemptive e non preemptive**: nel primo tipo è lo schedulatore che decide quando interrompere l'esecuzione del task corrente e sostituirlo con un altro, e questa azione di sospensione non volontaria è indicata col termine di preemption. Il tempo di esecuzione (timeslice) concesso a ciascun processo è generalmente predeterminato ed è proprio la capacità di manipolarlo che permette allo schedulatore di amministrare il funzionamento del sistema.

  Il secondo, indicato anche con il termine cooperativo, comporta che un processo non rilasci la cpu fino a quando non termina oppure non lo faccia volontariamente(*yielding*). In questo caso lo schedulatore ha una più limitata capacità di gestione, e questo può comportare scenari nei quali un unico task monopolizzi l'utilizzazione dell'unica cpu del sistema, non permettendo agli altri processi di proseguire ed essere completati.

- **Statici e dinamici**: i primi basano lo scheduling su parametri il cui valore rimane fisso nel tempo ed è assegnato ai job o ai task prima che questi siano attivati (release time), mentre nei secondi questi valori possono cambiare anche successivamente a tale istante.

- **Offline e online**: si riferiscono rispettivamente alla possibilità o meno di calcolare a priori la migliore strategia di schedulazione per l'intero sistema; nel caso degli algoritmi offline, ovviamente, è necessario conoscere in anticipo le caratteristiche temporali di tutti i task inclusi nel sistema in osservato.

- **Ottimali o euristici**: i primi sono quelli capaci di minimizzare una data funzione di costo definita sull'insieme dei task del sistema o che riescono a determinare una schedulazione fattibile se questa esiste; gli algoritmi euristici, invece, non garantiscono sempre la soluzione ottimale, ma solo un'approssimazione della stessa (vengono impiegati generalmente in problemi di tipo NP-Hard per determinare in tempi polinomiali una soluzione soddisfacente che tende verso quella ottima).

- **Guarantee-based o best-effort**: gli algoritmi guarantee-based operano garantendo sempre che, una volta selezionato per essere eseguito, un task terminerà la sua esecuzione entro la propria deadline e la totalità dei task accettati mantenga la propria fattibilità; per questo motivo, sono generalmente presenti nei sistemi di tipo hard real-time.

  Al contrario, i secondi operano in modo tale da soddisfare il maggior numero di deadline, senza però garantire la fattibilità della schedulazione; ottengono per contro migliori prestazioni medie rispetto ai precedenti e questo li rende adatti ad ambienti di tipo soft real-time.

Altra suddivisione [20] molto utilizzata è la seguente:

- **clock driven**: le decisioni riguardanti gli istanti di schedulazione e la durata di esecuzione di ciascun job sono definiti a priori e generalmente regolata da un qualche componente hardware che riveste il ruolo di temporizzatore. Applicati in sistemi con task periodici i cui parametri sono conosciuti a priori: questo consente di calcolare l'opportuna schedulazione anche off-line e riprodurla pedissequamente attraverso lo scheduler del sistema. Hanno il vantaggio di essere efficienti da eseguire e permettono una facile validazione del sistema, ma rsultano poco flessibili e qundi sostanzialmente inadatti ad ambienti soft real-time.

  Un tentativo di adattamento al soft real-time è stato fatto con lo *Slack Stealing*(Lehoczky e Ramos-Thuel nel 1992), nel quale se lo slack time non è nullo, si può utilizzare una prima porzione del frame,di dimensioni pari allo slack,per i processi soft real-time aperiodici, migliorandone quindi il gradi di responsività

- **round robin pesati**: caratterizzati dalla gestione dei job in modalità FIFO, dove ogni job viene eseguito per un intervallo massimo detto time slice. Se nella coda ci sono N job, un insieme di N time slice si dice round; per ogni round si avrà l'esecuzione di tutti i job. Se ai job vengono abbinati time-slice diversi determinati in base ad un parametro numerico detto peso, si parla appunto di round robin pesati. Anch'esso è di facile implementazione e garantisce una distribuzione equa del tempo cpu tra i job, ma non considera eventuali deadline dei job e risulta poco efficiente nella gestione di job con precedenze (esempio con 2 task, 2 job per task e 2 cpu dove $J_{1,1} < J_{1,2}$ e $J_{2,1} < J_{2,2}$ e $j_{1,1}$ e $J_{2,1}$ vincolati alla prima cpu e gli altri alla seconda: per il round-robin $J_{1,1}$ e $J_{2,1}$ sono eseguiti a timeslice e quindi gli altri due job vengono ritardati fino alla conclusione di $j_{1,1}$. Senza round robin $J_{1,1}$ sarebbe terminato per primo e in parallelo all'esecuzione di $J_{2,1}$ può essere attuata quella di $J_{1,2}$.

- **priority-driven**: caratterizzato dal non lasciare mai deliberatamente inutilizzato un processore o una risorsa (se c'è almeno un job presente, questo viene eseguito). Si realizza assegnando ai job una priorità e la schedulazione dipende quindi anche dall'ordinamento dei job in base alla priorità. Molti scheduler non real-time appartengono a questa categoria.

  La priorità può essere fissa, assegnata a ciascun task (ed ererditata dai suoi job) e non varia nel corso dell'esecuzione, o dinamica, che a sua volta si distingue per dinamicita a livello di task (i job già pronti per l'esecuzione non variano la loro priorità) o dinamici a livello di job (la priorità può cambiare anche dopo l'istante di release).

Riportiamo infine a titolo di esempio un breve e incompleto elenco dei principali algoritmi priority-driven presenti in letteratura, ampiamente trattati in [1] [3]:

- **FIFO(First in first out)**: la priorità dei job è proporzionale all'istante di arrivo.
- **LIFO(Last in first out)**: la priorità è assegnata in modo inversamente proporzionale al release time.
- **EDF(Earliest deadline first)**: la priorità del job è direttamente proporzionale alla vicinanza temporale della deadline.
- **LRT(Latest release time)**:
- **LST(Least slack time first)**:la priorità è inversamente proporzioinale allo slack del job.
- **SETF(short execution time first)**: la priorità è inversamente proporzionale al tempo di esecuzione.
- **LEFT(long execution time first)**: la priorità è direttamente proporzionale al tempo di esecuzione.
- **RM(rate monotonic)**: la priorità del job è direttamente proporzionale alla sua frequenza.
- **DM(deadline monotonic)**: priorità del job inversamente proporzionale alla sua deadlne relativa.

2.5 Sistemi embedded

Si tratta di sistemi programmabili per applicazioni specializzate, con una forte interazione con l'ambiente circostante. Generalmente o non sono dotati di interfaccia o, in caso contrario, essa risulta diversa da quella usuale di un calcolatore (tastiera, mouse e schermo). Sono sistemi pervasivi, diffusi all'interno di quasi ogni apparato elettronico.

Secondo le statistiche del World Trade Office [20], essi costituiscono il 98% dei dispositivi programmabili esistenti. Stimati in numero di 8 miliardi nel 2000, sono cresciuti fino a raggiungere i 16 miliardi nel 2010, con una proiezione di crescita che li vedrà toccare il numero di 40 miliardi nel 2020.

I settori in cui vengono maggiormente impiegati sono avionica, automotive, automazione industriale, telecomunicazioni, elettronica di consumo, domotica e apparati medicali; come si può ben vedere, esiste una forte sovrapposizione con i settori interessati dai sistemi di tipo real-time.

Gli usuali calcolatori vengono progettati primariamente per essere versatili, ovvero adattarsi al più ampio spettro di ambiti applicativi: di conseguenza, vengono dotati della maggior quantità possibile di risorse (sempre compatibilmente con il costo di produzione preventivato). Al contrario, i sistemi embedded devono usualmente svolgere un insieme ben definito di compiti, e questo consente di progettarli in maniera da rispettare particolari criteri di ottimizzazione (consumi, minima occupazione di risorse, ecc).

Generalmente, la loro progettazione deve tener conto di molteplici caratteristiche:

- Commerciali
  - Costo finale: incide sulle scelte di progetto
  - Time to market: il tempo necessario per presentare il dispositivo sul mercato, che non deve essere eccessivo in quanto si rischia di essere anticipati da altre aziende che operano nello stesso segmento.
  - Tempo di vita: ovvero la durata del dispositivo stesso, che può variare da pochi giorni a svariati decenni
  - Volume di produzione: legati ai costi, incidono specialmente sulla possibilità di recupero dei costi legati allo sviluppo del dispositivo
- Hardware
  - interfaccia di comunicazione: poichè il dispositivo in sè ha generalmente un costo basso, la scelta dell'interfaccia può influire pesantemente sul prezzo finale: si può andare da interfacce costituite da pochi pulsanti e led fino a interfacce grafiche complesse ed estremamente adattabili fornite su touch-screen.
  - consumo energetico: la gran parte dei dispositivi embedded è alimentata esclusivamente tramite batteria e quindi anche il fattore consumi riveste un peso rilevante nella determinazione delle specifiche di progetto.
  - dimensioni e peso: i dispositivi embedded devono essere solitamente compatti e leggeri (si pensi agli smartphone o ai riproduttori mp3) sia per garantirne l'ottima trasportabilità che per adattarsi ad essere inseriti in spazi di forma particolare.
- Software
  - Devono innanzitutto soddisfare le funzionalità tipiche alle quali la macchina deve sopperire.
  - Le dimensioni del codice sono generalmente contenute in quanto lo spazio di memorizzazione a disposizione è relativamente limitato.

  - qualità del servizio: le applicazioni possono dover soddisfare a particolari requisiti di qualità del servizio, sovente legati al supporto di processi in tempo reale.
  - aggiornabilità: consente di poter correggere eventuali errori anche in fase di post-produzione o di aggiungere ulteriori funzionalità.

- Di affidabilità
  - il sistema deve mantenere la probabilità di guasto entro limiti prefissati.
  - Deve essere definita un tempo massimo in cui il sistema può essere ripristinato o sostituito in seguito ad un guasto.
  - Se ne deve misurare la disponibilità, ossia il rapporto tra il tempo totale in cui il dispositivo risulta funzionante e il tempo globale in cui è attivo.
  - Il sistema deve essere sicuro, sia prevedendo la capacità di resistere a sollecitazioni o ambiti di utilizzo fuori dalle specifiche, sia impedendo che in seguito a guasti del sistema possano essere arrecati danni a cose o persone.

Tra i dispositivi embedded possiamo evidenziare la categoria di quelli che devono soddisfare a requisiti real-time, nei quali la reazione ad eventi esterni deve essere tempestiva e comunque contenuta entro specifiche temporali predefinite.

Un esempio è costituito dal dispositivo di controllo dell'airbag: se reagisce con troppo ritardo, oltre a non attutire l'impatto, rischia di arrecare ulteriore danno all'automobilista attraverso la successiva rapida espansione del cuscino; se invece opera con eccessivo anticipo, il cuscino ha il tempo di sgonfiarsi, annullando completamente lla sua funzione.

Il rispetto di tali tempistiche va attentamente valutato in sede di progetto; in questo caso, viene in aiuto dei progettisti la vasta base di studi e ricerche che vanno sotto il nome di teoria della schedulazione real-time e che comprendono appunto tutta una serie di algoritmi più o meno specifici in grado di garantire, in condizioni particolari, il completo rispetto dei vincoli temporali dell'insieme di processi che compongono il sistema in esame.

D'altro canto anche alcune componenti hardware vengono prodotte in funzione dell'ambiente in cui dovranno operare; emblematico è il caso di ARM[[20]] che con i modelli Cortex-R mira a progettare cpu ottimizzate per l'impiego specifico in ambito real-time.

Un ultimo aspetto da considerare riguarda la diffusione delle architetture multi-core [19], sia simmetriche che asimmetriche, nell'ambito dei dispositivi mobili. La loro integrazione comporta i seguenti vantaggi:

- **Riduzione del numero di componenti**: la miniaturizzazione consente di integrare in un unico package varie tipologie di processori, che

possono quindi accollarsi compiti prima eseguiti da componenti esterne, con conseguente miglioramento delle prestazioni (le comunicazioni tra le varie cpu sono più veloci) e un maggior contenimento dei consumi.

- **Funzionalità aggiuntive**: si possono integrare core dedicati ad eseguire elaborazioni specifiche, come ad esempio gpu o dsp.
- **Riduzione dei consumi**: la presenza di un numero elevato di core permette di avere più flussi elaborativi da sfruttare, senza dover necessariamente puntare sul solo incremento delle frequenze operative delle cpu per incrementare le prestazioni, incrementi la cui contropartita si realizza in consumi più che proporzionalmente maggiori.
- **Programmazione concorrente**: specialmente le applicazioni di tipo multimediale, caratterizzate da grandi quantità di dati da processare e vincoli temporali stringenti, possono sfruttare efficacemente questo tipo di archtetture suddividendo il carico elaborativo tra i core presenti; lo stesso sistema operativo che controlla il dispositivo ha la possibilità di prevedere meccanismi per determinare una ottimale distribuzione del carico computazionale tra questi.

D'altro canto, l'accresciuta complessità dell'hardware si riflette su un aumento delle problematiche da gestire a livello di sistema operativo: la sincronizzazione tra i vari flussi elborativi, l'accesso a risorse hardware condivise e la deterrminazione di interfacce che permettano di sfruttare la grande varietà di architetture multi-core mantenendo al contempo un insieme coerente ed uniforme di funzionalità da rendere disponibili per le applicazioni eseguite all'interno del sistema operativo stesso.

Capitolo 3

Il soft real-time

3.1 Sommario

In questo capitolo presenteremo in modo approfondito le problematiche legate al sistemi soft real-time, illustrando prima gli approcci alla loro gestione effettuati attraverso l'adattamento di sistemi general-purpose e hard real-time preesistenti, e scorrendo successivamente le principali soluzioni fornite dalla letteratura scientifica. L'intero capitolo è basato per la parte più consistente sul libro di Buttazzo [2].

3.2 Problematiche legate al soft real-time

Abbiamo già visto che nei sistemi soft real-time i vincoli temporali non devono essere rispettati in modo assoluto, poichè la violazione di una deadline non comporta conseguenze catastrofiche per il sistema e l'ambiente in cui è immerso ma si manifesta come semplice degrado prestazionale, da quantificare attraverso predeterminate tipologie di parametri di qualità del servizio (QoS).

Le ulteriori peculiarità di questa categoria di sistemi possono essere estrapolate dall'analisi di alcuni tipici scenari d'uso:

- **Estrema variabilità dei parametri temporali dei task**: ad esempio, in una riproduzione video, il tempo di decodifica di ciascun frame risulta dipendente dagli stessi dati contenuti in quest'ultimo, e di conseguenza il tempo di esecuzione del singolo job mostra un'estrema variabilità, come si può vedere anche dal grafico in Figura 3.1.

  Il tempo di esecuzione non è però l'unico; se pensiamo ad esempio al task che si occupa di rilevare la quota di un veivolo, esso presenta un periodo che si riduce al diminuire della quota stessa, in quanto la criticità del controllo di volo aumenta e necessità quindi di una maggiore frequenza nel rilevamento del dato per ottenere un quadro

della situazione quanto più possibile in tempo reale (si pensi ad esempio alle fasi di decollo e atterraggio).

- **Caratteristiche hardware della piattaforma**: sempre in relazione alla variabilità dei parametri temporali, il sistema deve tener conto anche della presenza o meno di meccanismi hardware quali il DMA, il prefetch o la cache[1] i quali, pur migliorando il comportamento medio del sistema (throughput), incidono negativamente sul grado di variabilità dei parametri temporali e quindi aumentano la variabilità dei tempi di risposta delle varie operazioni.

- **Dinamicità del sistema e dell'ambiente da controllare**: nei sistemi soft real-time, oltre ad avere generalmente una maggiore percentuale di task aperiodici o sporadici, i processi stessi possono essere creati o distrutti in qualsiasi momento e quindi la fattibilità o meno della schedulazione deve essere monitorata con maggior frequenza e non può essere determinata a priori.

- **Scelta del metodo di valutazione della qualità del servizio**: il fatto che i task non siano obbligati a rispettare in modo assoluto le deadline associate non significa che risultino accettabili frequenze elevate di queste violazioni: se pensiamo ad esempio alla riproduzione di un file multimediale che richiede una sicronizzazione audio/video, il suo contenuto diverrebbe non più usufruibile se il livello di asincronia fosse tale da essere percepibile dall'utente per lassi di tempo significativi.

  Diventa quindi fondamentale trovare un equilibrio tra la qualità dei risultati generati e l'efficacia nell'utilizzazione delle risorse messe a disposizione dalla piattaforma su cui si opera.

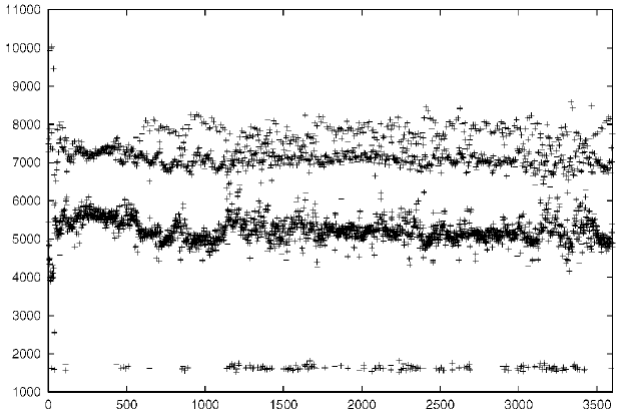


Figura 3.1: Tempi di decodifica per una sequenza di frame video (i tempi di decodifica sono espressi in microsecondi)

3.3 Primi approcci al soft real-time

I primi approcci alla gestione di questa tipologia di applicazioni hanno visto l'utilizzo di sistemi operativi già esitenti al fine di verificare se, malgrado fossero nati con scopi diversi, potessero comunque sopperire ai requisiti richiesti dal soft real-time. I risultati più significativi sono suddivisibili in due blocchi principali: quello derivante dall'impiego di sistemi operativi general purpose e quello legato ai sistemi hard real-time.

3.3.1 Sistemi general purpose

I sistemi operativi di tipo general purpose sono progettati con meccanismi mirati a massimizzare il throughput del sistema e questo avviene generalmente a scapito del suo grado di prevedibilità.

Se analizzati dal punto di vista del supporto al soft real-time, possiamo evidenziare i seguenti aspetti:

- Le strutture dati usate per gestire i task non prevedono il supporto per memorizzarne i vincoli temporali.
- Alcune funzioni come la delay, introdotta per posticipare l'esecuzione del codice di un ritardo prefissato, non sono sufficientemente affidabili, in quanto nella realtà garantiscono che l'esecuzione avverrà con un ritardo almeno pari, e non esattamente uguale, a quello definito.
- La sincronizzazione dei task può introdurre ritardi non limitati, come precedentemente visto nel caso di inversione di priorità .
- Si verifica una forte interferenza temporale tra i vari task del sistema; i processi ad alta priorità sono sempre privilegiati e, nel caso si verifichi in questi una situazione di overrun, il ritardo accumulato si ripercuote su tutti i task attivi a minore priorità, con il pericolo di innescare una violazione a cascata delle deadline associate.
- Alcune primitive dedicate allo scambio di messaggi utilizzano una semantica bloccante (nei casi si invii un messaggio ad una coda piena o lo si attenda da una coda vuota) e quindi il processo può rimanere fermo per un intervallo di tempo non limitato superiormente.
- Generalmente al meccanismo di gestione delle interruzioni (sezione 4.7.1) viene riservata la massima priorità; essendo gli eventi di interruzione asincroni rispetto al sistema e dunque in grado di manifestarsi con tempi o frequenze non controllati, possono introdurre ritardi di lunghezza arbitraria.

3.3.2 Sistemi hard real-time

L'elaborazione real-time è stata inizialmente introdotta per supportare sistemi critici per la sicurezza, come apparecchiature militari, controllo di volo, gestione di impianti nucleari o industriali, che successivamente sono stati indicati come sistemi hard real-time. In questi scenari l'apparato informatico deve garantire in qualsiasi condizione operativa di poter rispondere ad eventi esterni, provenienti dall'ambiente controllato, entro un intervallo di tempo fissato in sede di progetto. Per ottenere tali prestazioni, risulta fondamentale curare i seguenti aspetti:

- Determinare i massimi tempi di esecuzione dei task operanti all'interno del sistema. In questo modo si stabilisce un limite superiore (approccio di tipo pessimistico) al valore dei parametri temporali da usare come riferimento in quanto la priorità del sistema è garantire il rispetto dei vincoli di tempo. Le metodologie di misurazione impiegate possono essere di tipo sperimentale, ricavando il tempo impiegato per l'esecuzione di ciascun task su un range di condizioni iniziali il più ampio possibile, o analitico, studiando il codice del programma e individuandone le sequenze di esecuzione più lunghe, di cui poi si stima il tempo necessario per elaborarle. Entrambi gli approcci presentano dei punti deboli: il primo non può coprire l'intera gamma delle possibili condizioni di funzionamento (anche per motivi di costi), mentre il secondo deve fare delle assunzioni sulle tecnologie usate dalla cpu per eseguire il codice, in quanto meccanismi come il DMA, la cache o il prefetch, hanno un comportamento temporale largamente impredicibile, in quanto il tempo di esecuzione viene a dipendere, oltre dalla quantità di dati da trattare , anche dalla particolare sequenza delle elaborazioni precedenti.

- Limitare quanto più possibile i fattori di imprevedibilità del sistema. Ad esempiio, si utilizza generalmente una allocazione statica delle risorse, in modo che ogni processo, una volta in esecuzione, disponga a priori di tutte le componenti necessarie alla sua esecuzione; così facendo non si incorre nelle problematiche di imprevedibilità dei tempi di risposta dovute alla sincronizzazione.

- Una volta fissati i valori dei parametri, l'analisi di fattibilità e schedulabilità del sistema si appoggia a tutta una serie di algoritmi proposti in letteratura; in questa fase, per rendere più agevoli e rapidi i test, si introducono ulteriori restrizioni che accentuano ancora di più il carattere pessimistico delle ipotesi di base adottate.

L'applicazione di una tale metodologia risulta quindi adatta per sistemi relativamente semplici, caratterizzati da un comportamento deterministico e con parametri temporali stabili nel tempo o comunque con ristretti range di variabilità.

Lo stesso approccio, se applicato ad un ambiente soft real-time, mostra immediatamente una serie di forti limitazioni:

- Ll'adozione dei valori al caso peggiore dei parametri temporali comporta un forte spreco delle capacità elaborative. Torniamo all'esempio della riproduzione multimediale e poniamo l'ipotesi che la differenza tra il valori minimo e massimo nei tempi di decodifica di un frame abbia un rapporto di 1 a 10 e che però i valori massimi si presentino con una frequenza molto bassa; tarando il sistema sull'obiettivo di rispettare comunque i vincoli temporali, questo verrà sfruttato per la maggior parte del tempo per appena un decimo della sua potenza elaborativa.

  In aggiunta, l'allocazione statica delle risorse impone, a parità di task gestiti, la necessità di prevedere un maggior quantitativo delle stesse, con un conseguente innalzamento dei costi del prodotto finale. Questo aspetto riveste particolare rilevanza nei dispositivi embedded, dove capacità elaborativa e disponibilità di risorse sono relativamente limitate.

- Rispetto al caso precedente, non è nemmeno possibile ignorare i picchi dei tempi di esecuzione (legati ad esempio a fenomeni di overrun) solamente con l'obiettivo di incrementare lo sfruttamento delle risorse disponibili; se non opportunamente controllati, infatti, potrebbero indurre violazioni delle deadline in task a maggiore criticità.Possibili rimedi potrebbero essere la terminazione del task che presenta overrun o un abbassamento della sua priorità di esecuzione.

  Una tecnica adottata per arginare il fenomeno è quella della *resource reservation*, dove ad ogni task è riservata solo una frazione delle risorse disponibili attraverso la supervisione di un server che ne impedisce l'allocazione di un quantitativo superiore a quello stabilito. Questo approccio, però, risulta meno efficace in ambienti dinamici poiché le sue prestazioni dipendono fortemente dai valori dei parametri che regolano la distribuzione delle risorse; se inferiori alle necessità reali dei task, infatti, causano un allungamento dei tempi di esecuzione degli stessi, mentre se troppo elevati comportano uno spreco delle risorse disponibili.

- Scarsa adattabilità: gli algoritmi proposti in campo hard real-time vedono un degrado prestazionale più o meno marcato quando sono applicati ad un ambiente fortemente dinamico.

3.4 Peculiarità di un sistema soft real-time

Dall'analisi dei punti precedenti, e richiamando alcune delle caratteristiche proprie dei sistemi embedded, si possono far emergere le caratteristiche che

un sistema operativo soft real-time dovrebbe presentare:

- **Controllo delle situazioni di overload**: in modo che le richieste del sistema siano riportate gradualmente sotto una certa soglia per non ingenerare un crescente decadimento delle prestazioni.
- **Isolamento temporale**: in grado di garantire che situazioni anomale (generalmente overrun) relative ad un task influenzino solamente i suoi tempi di esecuzione e non quelli di tutti gli altri processi presenti nel sistema.
- **Supporto al multithread**: in questo modo la progettazione delle applicazioni può avvantaggiarsi della programmazione concorrente e della presenza di più core all'interno dello stesso dispositivo.
- **Controllo dell'inversione di priorità**: limitandone con opportuni meccanismi i ritardi indotti o altrimenti garantendo l'impossibilità che essa possa verificarsi.
- **Adattabilità**, in modo che i parametri di gestione del sistema si adeguino prontamente ai cambiamenti nei parametri relativi ai fenomeni controllati
- **Gestione degli eventi asincroni**: si garantiscono così i minimi tempi di risposta senza influire sui task periodici presenti, usualmente di maggiore importanza per il corretto funzionamento del sistema.
- **Inclusione di un meccanismo di recupero delle risorse**: si consente in questo modo di utilizzare ad esempio il tempo cpu rimanente dall'esecuzione di task terminati prima del previsto per sopperire ad eventuali situazioni di overload o per ridurre ulteriormente la latenza di risposta ai segnali asincroni.

3.5 Overload

I metodi proposti in letteratura per gestire l'overload sono molteplici e ne presenteremo quindi solo una visione generale. Essi si dividono sostanzialmente in due principali sottogruppi:

1. **Controllo di ammissibilità**: a ciascun task è associato un parametro che ne indica il grado di importanza; nel caso di possibile overload, i task meno importanti vengono completamente ignorati in modo da mantenere il carico del sistema sotto una soglia prestabilita, mentre i task rimanenti vengono serviti normalmente.

2. **Degrado delle prestazioni**: con questa politica nessun task viene rifiutato, ma per mantenere limitato il carico di lavoro è possibile che i processi vengano eseguiti con requisiti prestazionali inferiori a quelli inizialmente fissati.

3.5.1 Controllo di ammissibilità

Quando il sistema è sotto la soglia di massimo carico e viene impedita la creazione di nuovi task, non è necessario considerare l'importanza di questi ultimi, in quanto esistono algoritmi di schedulazione, quali l'EDF, ottimi nel caso di task prelazionabili, indipendenti e ovviamente in assenza di overload.

Nel caso di overload, invece, non esistono algoritmi certificati in grado di garantire una schedulazione ottimale, e quindi risulta importante distinguere tra vincoli temporali e importanza di un task.

Nei sistemi real-time, e in particolar modo in quelli di tipo soft, l'importanza corrente di un task può dipendere dall'istante in cui esso viene completato, e per questo si preferisce definire questa relazione tramite la funzione di utilità.

Una volta definite le funzioni da applicare, si può, in ogni momento, calcolare l'utilità globale del sistema come somma delle rispettive utilità dei job attivi nell'istante considerato. Su questo ulteriore parametro si possono sviluppare delle politiche di accettazione dei task che mirino ad esempio a massimizzarne il valore.

Considerato però che per ottenere il massimo valore possibile l'algoritmo dovrebbe conoscere in anticipo l'intera evoluzione dei task del sistema (compresa quella futura), al più le varie politiche potranno essere valutate più o meno positivamente a seconda dello scostamento del livello prestazionale raggiunto rispetto al valore ideale.

I vari approcci suggeriti per la determinazione di quali processi rigettare e quali accettare sono partizionabili in tre blocchi:

1. **Best effort scheduling**: non viene effettuato nessun controllo sui task in attesa di essere eseguiti e le prestazioni del sistema dipendono solamente dalla modalità scelta per l'attribuzione delle priorità.

2. **Simple admission control**: i nuovi task in attesa vengono controllati da un algoritmo di test e accettati solo nell'eventualità che il loro tempo di esecuzione, stimato al caso peggiore, garantisca ancora la schedulabilità dell'insieme dei processi correnti.

3. **Robust scheduling**: questa classe di algoritmi è un'evoluzione della precedente, in quanto nel caso l'accettazione di un task comporti una mancata schedulabilità esso non verrà rigettato ma si adotterà un secondo algoritmo per decidere quali dei task attualmente presenti eliminare. Questa modalità può trarre vantaggio da possibili politiche di

recupero del tempo cpu, collocando i processi rigettati in code temporanee per poterli riportare in esecuzione nel caso in cui i task correnti presentino tempi di esecuzione inferiori a quelli preventivati.

3.5.2 Degrado prestazionale

Anche in questo caso sono riconoscibili diversi approcci, ciascuno applicabile a contesti che devono presentare specifiche caratteristiche. Nell'elenco successivo ne forniremo alcuni esempi:

- **Service adaptation**: In alcuni contesti è possibile, nei casi non si disponga delle risorse necessarie per ottenere i risultati voluti nel rispetto delle deadline prefissate, ridurre il grado di accuratezza degli output forniti e quindi diminuire il tempo di esecuzione necessario per produrli.

  In questi frangenti, ciascun task viene suddiviso in una parte M obbligatoria e una O opzionale, dove la prima è quella necessaria ad ottenere un risultato di qualità minimale mentre la seconda ha lo scopo di raffinarlo ulteriormente ma può essere eseguita in modo parziale o completamente ignorata se ciò risulta necessario per soddisfare i vincoli temporali imposti.

  I due sotto-task così creati possiedono avviamente lo stesso release time ma sono legati dal vincolo che la parte opzionale potrà essere eseguita solo dopo il completamento di quella obbligatoria.

  Per misurarne le prestazioni, si introduce una grandezza detta errore valutata in base alla parte di task opzionale non eseguita; sommando in modo pesato l'errore di ciascun task si ottiene una valutazione quantitativa della precisione del sistema, mirata a mantenere la sommatoria il più prossima possibile al valore nullo.

  In queste condizioni, è possibile introdurre la distinzione tra *schedulazione fattibile*, ovvero capace di completare entro le deadline le parti obbligatorie dei vari processi, e *schedulazione precisa*, nella quale anche i sotto-task opzionali sono completati nei tempi richiesti.

  Nel caso non sia possibile suddividere i task nelle due porzioni indicate, l'algoritmo di service adaptation è ancora applicabile se si forniscono, per ciascun task, versioni diversificate nei tempi di esecuzione e nella precisione raggiunta; sarà compito dell'algoritmo di scheduling scegliere di volta in volta la tipologia offerta che meglio si adatta alle condizioni correnti del sistema.

- **Job skipping**:La classica schedulazione real-time assume che i processi periodici siano di tipo hard e quindi tutti i loro job debbano essere completati entro le associate deadline.

Un tale modello risulta troppo restrittivo in applicazioni quali le comunicazioni multimediali, nelle quali la mancata elaborazione di una certa percentuale di frame dati è tollerabile e può essere impiegata per contenere situazioni di overload temporaneo.

Il modello di job skipping, proposto originariamente da Koren e Shasha nel 1995, prevede che ogni task periodico sia caratterizzato da un massimo tempo di esecuzione, un periodo e un parametro s che indichi la distanza minima tra due successivi job scartati; ad esempio, se s è pari a 7 il sistema potrà eliminare al massimo un job ogni 7.

Nel loro modello essi dividono i job in rossi e blu, dove i primi vanno completati prima della deadline abbinata, mentre i secondi possono essere terminati in qualsiasi istante. Da queste premesse i due ricercatori hanno poi elaborato un paio di algoritmi di schedulazione: il *Red Task Only*, che rifiuta sempre i job blu e si occupa di schedulare quelli rossi con l'algoritmo EDF, e il *Blue When Possible*, che rappresenta un miglioramento del precedente in quanto mantiene lo stesso comportamento nei confronti dei job rossi ma include l'esecuzione anche quelli blu nel caso non siano presenti job rossi pronti per l'esecuzione.

- **Period adaptation**:In questo caso si agisce sul parametro che individua la periodicità di un task, variandola a seconda del carico di sistema.

  Questo approccio può risultare vincente nei sistemi multimediali, dove attività come il campionamento dei suoni, la compressione dei dati o la riproduzione di flussi video sono eseguite periodicamente ma la frequenza di esecuzione può essere variata entro un certo intervallo e garantire comunque una sufficiente qualità del servizio offerto.

  Gli approcci al problema sono molteplici; per questione di tempo illustreremo in modo generale solo il modello elastico presentato da Buttazzo e altri ricercatori nel 1998 e successivamente esteso nel 2002.

  Ciascun task viene modellato come una molla avente un determinato coefficiente di rigidità e vincoli di lunghezza; in particolare, l'utilizzazione legata a ciascun task viene trattata come un parametro elastico, modificato attraverso la variazione entro un certo range consentito del periodo del task stesso.

  Ad ogni task viene quindi associato un tempo di esecuzione, un periodo minimo ed uno massimo, e un coefficiente che, come la rigidità per la molla, descrive il grado di flessibilità con la quale il task può variare l'utilizzazione a lui connessa.

  Il problema di mantenere l'utilizzazione totale al di sotto di una certa soglia viene quindi traslato in quello di comprimere un certo numero di molle allineate per rimanere all'interno di una data estensione. La

soluzione raggiunta tiene conto anche di eventuali limiti inferiori al valore di utilizzazione abbinato ad un determinato task, che nel modello divengono molle che permettono una lunghezza minima prestabilita oltre la quale non possono essere compresse.

I vantaggi maggiori di questo approccio consistono nel fatto che la politica di schedulazione dei processi è direttamente legata ai coefficienti di elasticità che il sistema attribuisce ai vari task; inoltre, esso si adatta perfettamente nei sistemi multimediali nei quali la frequenza di esecuzione delle attività può essere opportunamente tarata in base al carico attuale del sistema.

Allo stesso modo, mantenendo aggiornato il valore dell'utilizzazione del sistema, se questa scende sotto il valore massimo impostato lo schedulatore può andare ad aumentare la frequenza di esecuzione di alcuni task, mantenendo l'intero apparato sempre in prossimità della massima utilizzazione e quindi realizzando uno sfruttamento ottimale delle risorse computazionali presenti.

3.6 Overrun e protezione temporale

Diversamente dal precedente aspetto, legato alla presenza permanente di possibili condizioni di overload, il meccamisno di protezione temporale va a mitigare gli effetti di overrun temporanei, isolando il problema al singolo task che ha presentato l'overrun ed evitando la diffusione degli effetti negativi all'intero sistema.

Una prima problematica legata all'overrun riguarda l'aumento del tempo di esecuzione di uno o alcuni job che compongono il task; poiché l'utilizzazione del task è pari a C/T con C tempo di esecuzione e T periodo del task, questa va ad aumentare, con la possibilità di far superare il valore dell'unità all'utilizzazione globale del sistema e far perdere il requisito di schedulabilità ai task presenti.

In secondo luogo, si può osservare che il verificarsi di una situazione di overrun in un processo di una data priorità influenzerà solamente i task di pari o inferiore prioprità. Questo, nei sistemi a priorità fissa, potrebbe indurre a gestire l'overrun tramite la manipolazione diretta delle priorità, ma questo approccio risulta generalmente inefficiente, specie in presenza di un elevato numero di processi attivi all'interno del sistema.

La protezione temporale si attua quando il comportamento di un task non viene in alcun modo influenzato da quello degli altri processi presenti nel sistema. Quando viene realizzata, presenta i seguenti vantaggi:

- L'overrun influenza solamente il task in cui si manifesta

- Permette di suddividere a priori il tempo cpu tra i task, garantendo a ciascuno una sorta di isolamento dato dall'ammontare di capacità elaborativa del processore assegnatagli.

- Possono essere fornite garanzie temporali diverse per le varie tipologie di processi (hard, soft, ecc).

- Se applicato a server aperiodici, protegge i task di tipo hard dalle attività di tipo sporadico o aperiodico.

- Si possono individuare task in overrun permanente e gestirli con una delle tecniche viste per il controllo dell'overload (ad esempio eliminando il task o rilassandone i parametri temporali di esecuzione).

Le classi di algoritmi che garantiscono la protezione temporale sono principalmente due: il fair scheduling e la resource reservation.

3.6.1 Fair scheduling

Si basa su un modello teorico (GPS, *Generalized Porcessor Sharing*) dove, in un qualsiasi intervallo di tempo, ciascun processo attivo riceve una porzione di tempo cpu.

Come si può notare, tale concetto non è praticamente realizzabile: l'adattamento ad ogni possibile intervallo temporale presupporrebbe una suddivisibilità infinita del tempo cpu, il quale invece presenta una granularità minima oltre cui non si può andare, data dall'inverso della massima frequenza a cui il processore può operare. Gli algoritmi reali, quindi, si prefiggono l'obiettivo di approssimarne quanto più possibile il comportamento ideale. La qualità di tale approssimazione viene misurata generalmente tramite un parametro detto *lag*, associato a ciascun task e definito come la differenza tra il tempo di esecuzione assegnato al task dall'algoritmo reale e quello previsto dal modello teorico; più il lag sarà prossimo a zero, migliore sarà l'approssimazione.

Gli algoritmi presenti in questa classe si basano generalmente sulla suddivisione del tempo cpu in *quanti* temporali di dimensione fissata, all'interno dei quali può essere eseguito un solo task alla volta; con la riduzione della grandezza di tali porzioni di tempo ci si dovrebbe avvicinare alle prestazioni ideali ma, come vedremo nei paragrafi successivi, questo non è vero nella realtà, in quanto i tempi per sostituire un task con un altro nel passaggio tra due quanti consecutivi non sono nulli ma legato all'intervallo necessario per eseguire un cambio di contesto.

Nonostante gli algoritmi siano ideati attorno al quanto come elemento comune„ se ne possono individuare due principali categorie, suddivise a seconda delle modalità di assegnazione delle porzioni di tempo cpu ai task:

- **Proportional share scheduling**: ad ogni task è assegnato un certo peso w e la porzione di tempo cpu viene calcolata come rapporto tra questo peso e la somma dei pesi relativi ai processi attivi nel sistema.Questo approccio presenta uno svantaggio: se il numero di task attivi varia, cosa frequente nei sistemi soft real-time, muta anche la porzione assegnata e quindi non si ottiene più l'isolamento temporale.

  Per continuare a garantirlo, si deve forzatamente implementare un secondo algoritmo per l'accettazione dei nuovi task, che valuterà se la mutata suddivisione della risorsa cpu è ancora compatibile con i parametri temporalidei vari task.

- **P-fair scheduling**: ad ogni processo si assegna un peso w_i tale per cui $\sum w_i <= M$ con M pari al numero di processori del sistema. In questo caso il peso corrisponde al tempo assegnato al task ed inoltre la condizione per l'accettazione o meno di un nuovo task si riduce alla verifica della disuguaglianza riportata.

3.6.2 Resource reservation

I primi algoritmi di questa classe furono proposti con l'obiettivo di fornire ai task aperiodici soft real-time un tempo di risposta minore possibile evitando al contempo di provocare il mancato rispetto delle deadline nei processi di tipo hard.

Questi algoritmi vanno sotto il nome di server aperiodici *(aperiodic server)*, ulteriormente distinti a seconda che essi manipolino task a priorità fissa o variabile.

Prima di addentrarci nella loro trattazione, illustreremo il concetto di server su cui essi si basano, evidenziandone le proprietà e le differenze rispetto all'approccio del fair scheduling.

Server

Ogni server è caratterizzato da un budget Q_i e un periodo P_I e la porzione di processore riservata a ciascun server è pari a Q_i/P_I. Il comportamento del server è del tutto simile ad un processo hard real-time con un tempo massimo di esecuzione Q_i e un periodo P_i, e quindi la sua analisi ben si prestava ad essere eseguita con le già consolidate tecniche impiegate per i sistemi hard real-time.

Questi algoritmi offrono una protezione temporale associando a ciascun task un server e controllando che la somma delle utilizzazioni di tutti i server attivi, ciascuna pari a Q_i/P_I, non superi una soglia predefinita (che deve essere non superiore al'unità). In questo modo viene anche a cadere la distinzione tra task periodici o sporadici, in quanto ognuno viene comunque gestito da un server periodico.

Ritornando per un attimo alle tecniche di fair scheduling, vediamo che il loro obiettivo è di mantenere vicina a zero la differenza tra tempo di esecuzione reale e tempo allocato dall'algoritmo ideale; questo comporta la necessità di un "quanto" il più piccolo possibile e, di conseguenza, un aumento della frequenza dei cambi di contesto. Lo stesso quanto, inoltre, determina in modo univoco la granularità minima della schedulazione per l'intero sistema, che quindi risulta difficilmente adattabile alle differenti esigenze dei task: il quanto potrebbe risultare, nelle situazioni estreme, o troppo lungo (con un conseguente spreco di potenziale elaborativo della cpu) o troppo breve (con un maggiore overhead legato ai cambi di contesto).

Al contrario, ciascun server della resource reservation è definito da due parametri, dove il periodo definisce la granularità della allocazione e il budget dà il peso abbinato a ciascun processo; questo consente di fornire questi due parametri adattandoli in funzione delle esigenze di ciascun task, e tale vantaggio comporta mediamente, a parità di condizioni, una drastica riduzione della frequenza dei context switch rispetto al fair scheduling.

Le modalità operative di ciascun server possono essere riassunte in tre punti:

1. Una volta creato, ad un server sono assegnati i parametri (Q_i, P_i) e un budget variabile q inizialmente pari a Q
2. Mentre il server gestisce l'elaborazione di un job, q viene decrementato di una quantità pari al tempo speso nell'esecuzione.
3. Quando il valore di q si annulla, possono essere intraprese due azioni distinte:
   (a) **hard reservation**: il job viene bloccato ; in questo modo, viene meno la proprietà di *work conserving*, ovvero è possibile che la cpu sia in stato idle anche in presenza di task in stato ready.
   (b) **soft reservation** il job viene sospeso e gli viene assegnata la minima priorità possibile; in questo modo ne potrà essere ripresa l'esecuzione nel caso il processore giunga in stato idle. Una tale strategia mantiene per il sistema la proprietà di *work conserving* solo nel caso in cui tutti i server adottino questa seconda politica di comportamento.

CBS

Una possibile implementazione della *resource reservation* si concretizza nel CBS (*Constant bandwidth server*). Per presentarne il funzionamento va introdotto il concetto di server attivo: una server si dice attivo all'istante t se esiste un job $\tau_{i,j}$ servito tale che $r_{i,j} <= t < f_{i,J}$, dove $r_{i,j}$ e $f_{i,j}$ sono rispettivamente il release time e il finishing time; in caso contrario, il server è considerato in stato idle. Le fasi con cui il CBS opera sono le seguenti:

1. ogni server (Q^s, P^s) è caratterizzato da un budget corrente q^s, una utilizzazione $U^s = Q^s/P^s$ e una deadline $d_{k,s}$ inizialmente nulla.

2. ad ogni job servito $\tau_{i,j}$ è assegnata una deadline $d_{i,j}$ pari a quella corrente del server.

3. come anticipato, durante l'esecuzione del job viene decrementato il budget q^s

4. quando $q^s = 0$, esso viene riportato al valore massimo Q^s e viene calcolata una nuova deadline $d^s_{k+1} = d^s_k + P^s$

5. se il job arriva quando il server è attivo, viene salvato in una coda di job pendenti gestita secondo una qualche politica

6. se invece il job $\tau_{i,j}$ arriva quando il server è in idle, se risulta soddisfatta la diseguazione $q^s >= (d^s_k - r_{i,j}) * U^s)$ (ovvero il server dispone di una banda insufficiente a garantire l'erogazione del budget corrente) viene ripristinato il budget al massimo valore e generata una nuova deadline, pari stavolta a $d^s_{k+1} = r_{i,j} + P^s$. altrimenti il job viene servito conservando la deadline e il budget corrente.

7. quando un job termina, se ci sono job pendenti questi vengono serviti, altrimenti il job passa allo stato idle.

8. in ogni istante, al job è assegnata l'ultima deadline generata dal server.

Il principale vantaggio di questo algoritmo risiede nel garantire la protezione temporale anche in situazioni in cui altri meccansmi falliscono; vediamone un esempio riferendoci alla Figura 3.2 e alla Figura 3.3. Abbiamo due task periodici $\tau_1 = (4, 8)$ e $\tau_2 = (3, 6)$ ciascuno gestito da un server tradizionale $S_1 = (4, 8)$ e $S_2 = (3, 6)$. Utilizzando l'algoritmo EDF per implementare la *resource reservation* non dovremmo riscontrare problemi, visto che l'utilizzazione totale è pari a $U = 4/8 + 3/6 = 1$ e quindi i task costituiscono un insieme schedulabile con l'algoritmo scelto.

Il ritardo con cui uno dei job di τ_1 si presenta, però, comporta la violazione della deadline di un job del secondo task e si perde così la proprietà di protezione temporale.

Nella stessa situazione, grazie all'assegnazione di priorità dinamiche a ciascun job, il CBS opera nel modo corretto: il job che presnta il ritardo giunge infatti con il proprio server in stato idle e va quindi verificata all'istante 18 l'equazione riportata al punto 6; essendo verificata $(4 >= (24 - 18) * (4/8))$ viene associata al job una nuova deadline $d_{k+1} = 18 + 8 = 26$ e l'EDF va a dare la precedenza al job di τ_2, isolando gli effetti negativi del ritardo all'interno del task in cui questo si è verificato.

Oltre alla protezione temporale, il CBS rivela altre proprietà interessanti:

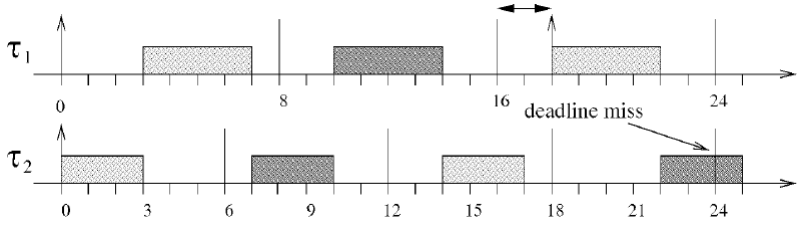


Figura 3.2: EDF fallisce nel garantire la protezione temporale

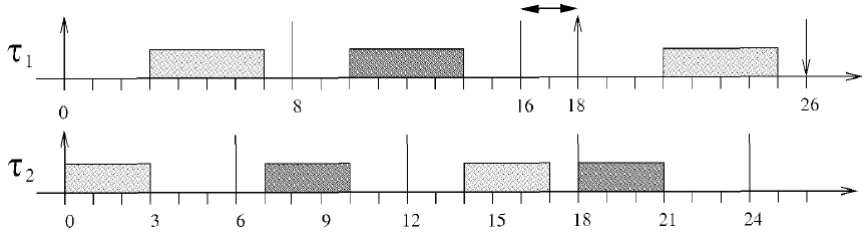


Figura 3.3: CBS garantisce la protezione temporale anche con arrivi ritardati.

- non serve fare alcuna ipotesi sui tempi di esecuzione massimi dei task o sugli intervalli di inter-arrivo, realizzando un disaccoppiamento tra tipologia di task presenti e parametri di schedulazione.
- conoscnedo la distribuzione statistica del tempo di esecuzione dei job di un task se ne possono soddisfare garanzie temporali espresse in termini di probabilità.
- viene recuperato qualsiasi intervallo di elaborazione reso disponibile da esecuzioni terminate prima del previsto o arrivi ritardati; questo grazie all'immediato ripristino del budget corrente non appena questo diviene nullo è alle modalità di posticipazione della deadline.

Le proprietà [9] sono però mantenute solo nel caso di processi indipendenti e in assenza di condivisione di risorse in mutua esclusione.

3.7 Il multi-thread

Il modello di programmazione concorrente usato per lo sviluppo delle applicazioni può risultare vincente in ambienti con scarse risorse in quanto i thread, a differenza dei processi, godono di numerosi vantaggi: possono condividere lo stesso spazio di indirizzamento e altre risorse, comunicano attraverso strutture dati condivise, sia la loro creazione che il cambio di contesto fra thread risultano meno dispendiosi e quindi più rapidi da elaborare e infine consentono generalmente una migliore utilizzazione dei sistemi multi-core.

Per contro, nei sistemi che devono gestire applicazioni multiple, ogni processo è sviluppato indipendentemente e quindi necessita di meccanismi di protezione per evitare interferenze con gli altri task presenti, mentre i thread sono implicitamente caratterizzati da una forte interazione, tanto che un malfunzionamento di uno solo di essi può compromettere l'intera applicazione.

Per creare un ambiente ottimale all'esecuzione di applicazioni soft real-time multithread dovrebbero quindi essere soddisfatte dal sistema le seguenti proprietà:

- deve essere possibile assegnare a ciascuna applicazione una frazione delle risorse di sistema, in modo da essere eseguita in una sorta di cpu virtuale.
- il tempo di esecuzione dovrebbe essere attribuito in modo opportuno a seconda dei requisiti real-time della stessa.
- le applicazioni non real-time non dovrebbero influire sulle garanzie temporali offerte a quelle real-time.
- possibilità di gestire almeno un doppio livello di schedulazione: una *globale* relativa alle applicazioni presenti, ed una *locale* a ciascun processo. In questo modo, ogni applicazione può essere realizzata con l'algoritmo di scheduling più adatto alle proprie caratteristiche e se ne incrementa la protabilità su vari sistemi, in quanto essa rimane indipendente dalla particolare tecnica di schedulazione globale adottata.

L'elenco ricorda da vicino le proprietà ottenibili con la *resource reservation*, mirata però all'intera applicazione e non al singolo thread che la compone; si realizzerebbe così la protezione temporale dei processi presenti e la possibilità quindi di effettuare l'analisi di schedulabilità indipendentemente per ciascuna applicazione.

Purtroppo, come visto precedentemente, non è possibile ottenere una perfetta protezione temporale per via della non infinita divisibilità del tempo cpu; ad esempio, nel caso del CBS, la periodicità del server è indice della granularità temporale, e valori troppo elevato o troppo bassi della stessa comportano i problemi già visti alla sezione 3.6.2.

Le problematiche aggiuntive legate all'ultimo punto (schedulazione a due livelli) si riassumono in una sequenza di tre passi: scegliere l'algoritmo di scheduling globale, calcolare come suddividere il tempo cpu per rispettare i vincoli temporali delle varie applicazioni e capire in che modo effettuare l'analisi di schedulabilità.

3.7.1 Algoritmo di Deng e Liu

È il primo in cui si considera il concetto di *sistema aperto*, tale per cui non si ha conoscenza a priori delle applicazioni (in questo caso manteniamo

distinto il concetto di applicazione dai concetti di processo, task o thread che invece sono ritenuti sinonimi) che saranno attive nel sistema in ogni istante e dove di conseguenza non si possono effettuare analisi di schedulabilità off-line ma occorre invece poter validare il singolo task inidipendentemente dal comportamento del resto del sistema.

Il modello distingue tre diversi tipi di processi, associando a ciascuno distinte garanzie temporali e algoritmi di schedulazione:

- **Applicazioni non prelazionabili**: all'interno della stessa applicazione i thread non possono inrterrompersi vicendevolmente, mentre possono subire preemption da thread di altre aplicazioni.
- **Applicazioni prelazionabili e predicibili**: è consentita la mutua interrompibilità ma si conoscono a priori gli istanti in cui interviene lo scheduler locale.
- **Applicazioni prelazionabili e non predicibili**:i thread possono interrompersi a vicenda e inoltre la presenza di thread aperiodici o sporadici rende imprevedibili gli istanti di azione dello scheduler.

Di base l'algoritmo reagisce all'arrivo di una nuova applicazione, la quale dichiara il tempo di esecuzione richiesto tramite il parametro c_i, assegnandogli un server S_i di utilizzazione massima U_i, in modo tale che l'utilizzazione totale di sistema non superi l'unità (lo scheduler globale è l'EDF). Oltre ai soliti parametri, ogni server include anche un parametro e_i detta *eleggibilità*; un server risulterà eleggibile (ovvero potenzialmente selezionabile dallo scheduler globale) all'istante t se $t \geq e_i$.

Esso poi si differenzia nel comportamento a seconda di quale delle precedenti tipologie di applicazione deve andare a gestire. Nel caso ad esempio delle applicazioni non prelazionabili opera secondo le seguenti regole:

- se all'istante t arriva l'applicazione A_i, se la coda dei thread serviti dallo scheduler locale è vuota ed esso è eleggibile, allora si impostano $q_i = c_i$ e $d_i = \max t, d_i + \frac{c_i}{U_i}$ (equivalenti alle equazioni già viste per il CBS) e il server viene inserito nella coda globale dell'EDF; in caso contrario (coda non vuota o server non eleggibile), è semplicemente inserito nella coda dello scheduler locale
- se un server è selezionato da EDF per essere eseguito, inizia ad elaborare il primo thread della sua coda e decrementa di pari passo il proprio budget.
- il server opera fino al termine del thread o all'esaurimento del budget; nel primo caso, si pone $e_i = d_i$ e il server diviene non eleggibile, mentre nel secondo si genera un'eccezione con la quale si comunica all'applicazione A_i di operare l'azione appropriata in risposta a aquesto evento (l'algoritmo lascia la decisione allo sviluppatore)

L'algoritmo gode dell'importante proprietà che, se una applicazione non prelazionabile è schedulabile in un processore dedicato di utilizzazione U_i, allora lo è anche con il server, definito dall'algoritmo, avente lo stesso valore di utilità.

Il suo limite però risiede nel poter gestire solamente applicazioni in cui lo scheduler globale deve essere non prelazionabile.

Per poterlo superare, gli autori hanno individuato le altre due classi di applicazioni e, in base alle proprietà intrinseche delle stesse, hanno modificato l'algoritmo in modo da mantenere la corrispondenza tra utilizzazione del server assegnato e del processore dedicato. Nell'ultimo caso, essendo tutti i parametri sconosciuti (eccetto il tempo di esecuzione c_i), si usa un quanto ϵ per mantenere aggiornato il budget e si introduce quindi un margine di errore tra la corrispondenza delle due utilizzazioni (al diminuire di ϵ si riduce anche l'errore, a prezzo però di un maggiore overhead dell'algoritmo stesso).

3.7.2 BSS

Anche qui ogni applicazione A_i è abbinata ad un server S_i che mantiene una coda, ordinata secondo lo scheduler locale, dei thread pronti all'esecuzione, con l'usuale ipotesi che la somma di tutte le U_A sia non superiore a 1.

Ogni qualvolta un thread dell'applicazione è pronto, il server S_i calcola il proprio budget e la propria deadline, andando ad inserirsi successivamente nella coda globale dell'EDF. Le due grandezze vengono così determinate:

- la deadline è pari a quella del thread di A_i con la deadline più prossima.
- il budget viene calcolato tramite una struttura interna ad ogni server detta *lista dei residui*: un residuo è una coppia $l = (B_a, d)$ con d deadline del thread e B_a budget disponibile nell'intervallo a, d con a release time del thread. La lista viene ordinata per deadline non decrescenti, e così devono risultare anche i budget assegnati

  Il residuo con la deadline più vicina viene infine assegnato al server.

Vengono definite poi alcune operazioni per gestire la lista dei residui:

- aggiunta: quando un nuovo thread dell'applicazione diviene quello con la deadline $d_{i,j}$più vicina tra quelli dell'applicazione, un nuovo residuo viene inserito nella lista ordinato secondo $d_{i,j}$ e con un budget [11]

  $$B_{i,j} = \min D_{i,j} * U_A, (d_{i,j} - d_{k-1}) * U_A + B_{k-1}, B_k$$

  dove $D_{i,j} = d_{i,j} - a_{i,j}$, d_{k-i} è la deadline del residuo che precede nella lista il nuovo entrato e B_{k-1}, B_k sono rispetivamente i budget del residuo precedente e successivo. Il motivo di una tale scelta è illustrato in Figura 3.4: la banda del nuovo residuo non può superare quella del successivo, definito in un intervallo temporale più ampio, e non può

superare per lo stesso motivo $(d_{i,j} - d_{k-1}) * U_A + B_{k-1}$; infine, non deve superare quella assegnata al server $(D_{i,j} * U_A)$ che stà gestendo l'applicazione.

- aggiornamento: effettuato ogni volta che un thread termina la sua esecuzione, viene prelazionato da un nuovo thread con deadline minore o viene esaurito il budget. Detto k il residuo corrispondente alla deadline attuale del server, il budget dei successivi viene decrementato di una quantità e pari al tempo di esecuzione del thread interrotto o completato, mentre i precedenti con budget superiore a quello del residuo in esame vengono rimossi.

- eliminazione: detto l_k il residuo in osservazione, esso viene eliminato se $d_k \leq t$ (t istante corrente) o se $B_k > (d_k - t) * U_A)$, ovvero se la banda associata supera quella disponibile nell'intervallo $[t, d_k]$. L'eliminazione è giustificata dal fatto che questi elementi non influiscono più sulla determinazione del budget. Infatti, se supponiamo che un nuovo residuo l_j sia inserito rspettivamente prima o dopo l_k, abbiamo che:

  - $D_j * U_A = (d_j - t) * U_A < (d_k - -t) * U_A < B_k$ dove la prima diseguaglianza segue dall'ordinamento delle deadline e la seconda dal fatto che $(d_k - t) * U_A < (d_k - a_k) * U_A = B_k$.
  - $D_j * U_A = (d_j - t) * U_A < B_k + (d_j - d_k) * U_A$, evidenziabile sempre dalla Figura 3.4 ponendo $d_{k-1} = d_k$ e $d_i = d_j$.

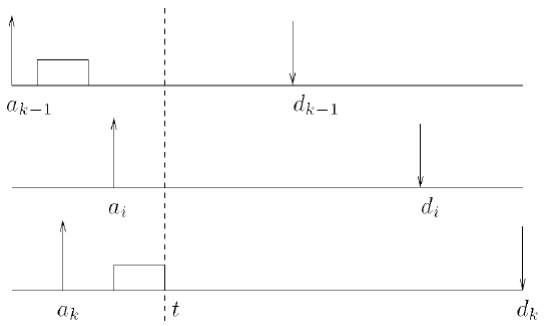


Figura 3.4: Calcolo del budget dei residui

L'algoritmo BSS mantiene la stessa proprietà di protezione temporale fornita dal CBS indipendentemente dal tipo di scheduling locali delle applicazioni e dai tempi di arrivo e di esecuzione dei thread che le realizzano.

Sia il BSS che l'algoritmo di Deng e Liu necessitano del tempo totale di esecuzione richiesto dall'intera applicazione e per questo sono classificati come *algoritmi intrusivi* (sfruttano informazioni temporali note a priori).

Questo ne esclude l'utilizzo nei contesti dove questo parametro non può essere preventivamente determinato; inoltre, esiste una forte interdipendenza tra lo scheduler globale e quello locale, mentre sarebbe utile poterne disaccoppiare completamente il comportamento.

Il prossimo modello mira esattamente a superare tali limitazioni.

3.7.3 Partizionamento delle risorse

L'algoritmo si fonda sul concetto di *partizione di risorsa* che, nella sua accezione più generale, corrisponde ad una funzione $\Pi(t)$ che può assumere solo due valori, o 0 o 1; se pari ad 1 la risorsa è allocata all'applicazione e può quindi essere utilizzata, altrimenti no.

Si definiscono poi:

- Fattore di disponibilità: è un valore $\alpha = \lim_{x\to\infty} \frac{\int_0^x \Pi(t)dt}{x}$
- Funzione di supporto: indicata con $S(t)$ e pari alla disponibilità della risorsa tra l'istante 0 e l'istante t.
- Least supply function: funzione $S^*(t)$ pari al $\min S(t+d) - S(t)$, con $d, t \geq 0$.
- Partizione critica: è una partizione di risorsa pari a 1 negli intervalli in cui $S^*(t)$ risulta strettamente crescente.
- Ritardo di partizione: quantità Δ pari al $\min d$ tale che $S^*(t) - \max \alpha * t - d, 0 \geq 0$.

La teoria sviluppata attorno a questo modello, oltre a garantire che se un task è schedulabile (con il particolare algoritmo locale) nella partizione critica lo è anche all'interno della partizione di risorsa da cui questa è stata ricavata, permette di stabilire una correlazione diretta tra la coppia di valori (α, Δ) del task e i parametri (Q_i, P_i) del server da associarvi.

Rispetto ai precedenti, quindi, il partizionamento delle risorse ha il vantaggio di consentire allo scheduler globale di garantire i vincoli delle applicazioni (determinando il server da associare a ciascuna), senza doverne conoscere i valori dei parametri temporali associati.

3.8 Sincronizzazione

Gli algoritmi di schedulazione visti precedentemente non tengono conto del fatto che i task possono auto-sospendersi o utilizzare delle primitive bloccanti (ovvero che, in attesa del completamento dell'azione richiesta, sono in grado di porre in stato di sleep il processo che ne ha richiesto l'intervento).

Nella realtà dei sistemi operativi, invece, questa situazione si verifica frequentemente, specialmente nelle operazioni di sincronizzazione per l'accesso a risorse condivise.

Come già anticipato, questa esigenza introduce varie problematiche nel comportamento del sistema, affrontate ampiamente in ambito hard real-time e alle quali sono state proposte soluzioni basate su alcuni algoritmi di base:

- **Priority Inheritance protocol (PIP)**:Quando un processo a priorità maggiore è bloccato all'ingresso di una sezione critica a causa di un task a minore priorità, quest'ultimo assume temporaneamente la medesima priorità del primo, tornando poi a quella originale una volta rilasciata la risorsa. L'applicazione di tale protocollo consente al task di rimanere bloccato al massimo per la durata della sezione critica di ogni task a priorità minore, indipendentemente dal numero di semafori che potenzialmente possono bloccarlo, o per la durata della sezione critica di ogni semaforo che può potenzialmente bloccarlo. In base a queste proprietà [9], un insieme di n task (hard), ordinati per periodi crescenti, per essere schedulabile deve verificare la seguente disequazione:

  $$\forall i \quad 1 \leq i \leq n \quad \sum_{j=1}^{i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq 1$$

  dove C_i e T_i sono rispettivamente il tempo di esecuzione al caso peggiore e il periodo del task τ_i e B_i è il massimo periodo di blocco ad esso abbinato.

  Il PIP riduce l'inversione di priorità ma non elimina tutti i problemi che ne derivano: nel caso infatti di sezioni critiche innestate, un processo a bassa priorità che ne blocca altri può continuamente ricevere alte priorità temporanee, bloccando al contempo processi su altre risorse detenute; in più, un processo può essere bloccato da molti task su risorse differenti (*chained blocking*) Inoltre, se non è imposto un ordine nell'acquisizione e rilascio delle risorse, il fenomeno del deadlock non viene minimamente intaccato.

- **Priority ceiling Protocol (PCP)**:Permette di migliorare il PIP assegnando a ciascuna risorsa un valore a priori detto ceiling e pari alla massima priorità dei task che potrebbero richiedere il lock della stessa. L'algoritmo si basa sulle seguenti regole:

  1. all'istante t, un task può eseguire il lock sulla risorsa R se la sua priorità è strettamente maggiore di tutti i valori di ceiling delle risorse attualmente assegnate ad altri job.
  2. Se un task è bloccato su R da un altro a priorità minore, quest'ultimo ne eredità la priorità come nel caso del PIP

3. Se un task T rilascia una risorsa e non vi sono altri task bloccati da questo sulla medesima risorsa, esso riprende la priorità originaria.In generale, quindi, un task opera o con la priorità originale o con la priorità massima dei task che stà bloccando.

Questo permette al PCP di scongiurare fenomeni quali il deadlock e i blocchi a catena che il PIP non riusciva a gestire; inoltre, un task può rimanere bloccato al massimo per la durata di una sezione critica. Ha però il limite di dover impostare a priori il valore di ceiling, cosa fattibile se si conoscono a monte tutti i task che verranno generati e le loro esigenze in fatto di risorse, e quindi poco adattabile ad un ambiente dinamico come quello del soft real-time.

Le soluzioni precedenti riguardano i sistemi di tipo hard real-time, e questo lo si può intuire dal fatto che gli algoritmi (il secondo in particolare) tendono ad usare almeno un parametro statico nelle loro elaborazioni.

In contesti soft real-time o misti il problema della sincronizzazione e della gestione delle problematiche ad essa legato è più complesso. Delle varie soluzioni presentate, concentreremo la nostra attenzione sull'algoritmo che combina CBS e PIP, in quanto non richiedendo conoscenze a priori sul comportamento dei task meglio si adatta ad essere impiegato in sistemi soft real-time.

3.8.1 BWI

Nel dettaglio [9], le condizioni dalle quali si è partiti per definire il nuovo algoritmo corrispondono alle seguenti:

- i tempi di arrivo di ciascun job non sono determinabili a priori.
- la stessa condizione vale per i tempi massimi di esecuzione, che possono essere ricavati solamente proseguendo l'elaborazione del job fino alla sua terminazione.
- l'algoritmo non dispone di alcuna indicazione su quali risorse del sistema saranno accedute da ciascun job.

La fusione di CBS e PIP comporta inoltre alcune questioni preliminari:

- risolvere il problema del deadlock, in quanto PIP non concorre ad eliminarlo.
- ridefinire il concetto di isolamento temporale quando si considerano task interagenti tramite risorse condivise in mutua esclusione, visto che nella trattazione del CBS abbiamo considerato esclusivamente processi totalmente indipendenti.

- determinare il tempo di blocco di ciascun task.

- stabilire come comportarsi nel caso il budget di un server si esaurisca proprio all'interno di una sezione critica.

Una possibile risposta viene offerta appunto con il protocollo *Bandwidth Inheritance* (BWI), basato sull'idea di far ereditare al job bloccante a minore prorità il budget corrente e la deadline del server bloccato.

In quanto al primo problema, viene risolto agevolmente imponendo un ordinamento per l'accesso alle risorse: più precisamente, se $i < j$ allora R_j sarà accessibile solamente all'interno di una sezione critica relativa ad R_i. in questo modo si eliminano possibili concatenazioni che formano un loop e danno luogo al deadlock.

Per quanto riguarda l'isolamento, si può osservare che il task τ_i può essere bloccato direttamente o indirettamente da τ_j. Viene definita la sequenza bloccante per τ_i come $BC_i = (\tau_1, R_1, \tau_2, R_2, ..., R_{z-1}, \tau_z)$ dove $\tau_i = \tau_1$, $\forall k = 1, ..., z-1, \tau_k$ e τ_{k+1} condividono R_k e, se $z > 2$ $\forall k = 2, ..., z-1, \tau_k$ accede a R_k a partire da una regione critica relativa a R_{k-1}.

Si crea quindi una concatenazione dove i task precedenti risultano sempre potenzialmente bloccabili dai successivi. Potendo esistere per ogni task più catene bloccanti, il massimo ritardo possibile sarà legato al tempo critico massimo di tutte le sequenze bloccanti a lui relative.

E veniamo infine alle modalità di funzionamento di BWI. Potendo lo stesso server farsi carico di più task simultaneamente, ad ognuno sarà associata una lista contenente tali task; definiamo inoltre la quantita $e(i,t)$ come l'indice del server con la deadline più vicina all'istante t contenente il task τ_i. Inizialmente, ogni server conterrà un solo task e quindi $e(i,0) = i$, chiamato anche *server di default* per il task τ_i. Per il resti BWI si comporterà alla stregua del server CBS con due regole aggiuntive:

- se τ_i è bloccato nell'accesso alla risorsa R dal task τ_j, quest'ultimo sarà aggiunto alla lista del server $S_{e(i,t)}$ e la cosa sarà ripetuta ricorsivamente fino a trovare il primo task non bloccato; in questo modo il server $S_{e(i,t)}$ conterrà l'intera catena bloccante relativa a τ_i.

- nel momento in cui τ_j rilascia la risorsa detenuta, se su questa sono presenti altri task bloccati significa che τ_j è stato completato in un server diverso da quello di default (in base alla regola precedente). Al server $S_{e(i,t)}$ non resta che eliminare τ_j dalla propria lista; così facendo τ_i sarà il nuovo task sbloccato e dovrà essere sostituito al posto di τ_j in tutte le liste in cui quest'ultimo è ancora presente.

L'algoritmo quindi fa ereditare al task τ_j bloccante a minore priorità i parametri del server $S_{e(i,t)}$, almeno fino a quando questo non esaurisce il suo budget; dopo di che, la deadline del server sarà posticipata e τ_j proseguirà

la sua esecuzione su un server $S_{e(j,t)}$ che potrebbe non corrispondere più al $S_{e(i,t)}$.

Il protocollo BWI gode delle seguenti proprietà fondamentali:

- ogni server attivo ha sempre uno e un solo task attivo nella sua lista.
- dato un sistema con n server tali per cui $\sum_{i=1}^{n} U_i \leq 1$ e che usano il protocollo BWI per l'accesso alle risorse condivise, nessun server viola le proprie deadline.

Si sottolinea che la seconda proprietà fa riferimento alle deadline usate dal server per schedulare un job con l'algoritmo EDF e non le deadline del job, che quindi possono eventualmente non essere rispettate.

3.9 Recupero delle risorse

Negli ambienti di tipo soft real-time il tempo di esecuzione dei job presenta una forte variabilità e quindi il loro comportamento non è definibile a priori in modo deterministico.

Le prestazioni del sistema garantite dalle tecniche di resource reservation viste soffrono da questo punto di vista vengono a dipendere fortemente da una corretta distribuzione delle risorse tra i vari processi: se queste risultano eccessivamente inferiori rispetto alle richieste medie del task questo presenterà una bassa responsività, mentre in caso contrario il sistema lavorerà in modo inefficiente, sprecando una parte della banda di capacità elaborativa.

L'idea alla base degli algoritmi che andremo a presentare risiede quindi nella capacità di recuperare le risorse inutilizzate dei server impiegati nell'implementazione della resource reservation, mettendole a disposizione per completare altri processi attivi presenti nel sistema.

3.9.1 CASH e GRUB

Entrambi gli algoritmi mirano ad instaurare un meccanismo di recupero delle risorse mantenendo al contempèo intatte le garanzie di isolamento temporale fornite dai server. Inoltre, la loro definizione si basa su alcune condizioni iniziali:

1. I task siano schedulati attraverso un assegnamento dinamico delle priotirà, come fatto ad esempio dall'algoritmo EDF.
2. I processi non devono presentare dipendenze di qualsiasi tipo. Questo vincolo è fortemente limitante in quanto esclude qualsiasi sistema in cui i task operano con risorse condivise in mutua esclusione.
3. ogni task viene gestito da un server dedicato.
4. Entrambi si fondano sull'algoritmo CBS.

CASH

Il meccanismo di recupero CASH (*Capacity Sharing*) è imperniato su una lista globale (*CASH list*) di elementi costituita da coppie di valori, nel cui il primo è relativo alla capacità residua messa a disposizione da un server S e il secondo corrisponde alla deadline del server stesso.

Non appena un nuovo task viene portato in esecuzione, il server inizialmente utilizza le capacità residue presenti nella lista che hanno deadline minori o uguali di quella attualmente assegnata al server. Una volta che un task termina, se ci sono task pendenti questi vengono serviti ancora con il budget e la deadline correnti, altrimenti il server si pone in stato idle e l'eventuale capacità residua viene inserita nella coda CASH.

Resta da evidenziare che, se l'intero sistema rimane in idle per un intervallo Δ, l'eventuale capacità residua con la deadline più prossima presente nella coda CASH viene decrementata di una uguale quantità.

I vantaggi più evidenti di questo algoritmo sono i seguenti:

- La condizione di schedulabilità (sempre in riferimento ad un sistema uniprocessore) rimane $\sum_{j=1}^{i} \frac{Q_j}{T_j} \leq 1$.
- Si ottiene una diminuzione della frequenza delle azioni di posticipo delle deadline dei server CBS e quindi un contenimento dei tempi di risposta dei task serviti e un più efficiente utilizzo delle risorse di calcolo.

Riferimenti a studi più recenti [21] indicano che ulteriori analisi sono state condotte per studiare il comportamento della lista di capacità residue adottata dall'algoritmo.

GRUB

il *Greedy reclamation of Unused bandwidth* (GRUB) mira a recuperare la larghezza di banda sprecata dai server che non hanno alcun job in attesa di esecuzione.

Per ciascun server, caratterizzato da un periodo P_i e una banda U_i, l'algoritmo conserva una variabile V_i (*virtual time*), usata per quantificare quanta della banda riservata al server è già stata utilizzata, e uno stato s_i che può assumere, al tempo t_0 uno dei seguenti tre valori: *activeContending (AC)* se possiede job in attesa di esecuzione, *activeNonContending (ANC)* se ha completato tutti i job rilasciati prima di t_0 ma ha già impiegato più della propria larghezza di banda concessagli fino a tale istante (ovvero ha $V_i > t_0$) e *inactive (I)* se non possiede job pendenti e non ha utilizzato la porzione di cpu oltre quella concessagli.

Inoltre, conserva una variabile globale $U(t)$ pari alla somma delle utilità di tutti i server attivi all'istante t.

L'algoritmo opera poi seguendo questi passi:

- se un job J_i arriva all'istante a_i in un server S_i in stato I, si pongono $V_i = a_i$ e $d_i = V_i + P_i$ e il server passa in stato AC
- se un job J_{i-1} termina su S_i, si hanno due possibili scelte:
  - se il job successivo J_i è già arrivato, si pospone la deadline con $d_i = V_i + P_i$ e il server rimane in stato AC
  - se non ci sono altri job su S_i e $V_i > t$ con t istante corrente, il server passa in stato ANC, mentre se $V_i \leq t$ allora si pone in stato I.
  - se un job J_i giunge su S_i mentre questo è in stato ANC, si aggiorna la deadline con $d_i = V_i + P_i$ e si attua la transizione allo stato AC.
  - mentre il server S_i è in esecuzione, la variabile V_i è incrementata secondo la seguente legge:

$$\frac{d}{dt}V_i = \begin{cases} \frac{U}{U_i} & \text{se } S_i \text{ è in esecuzione} \\ 0 & \text{altrimenti} \end{cases}$$

    Se V_i diviene pari a d_i, quest'ultima viene posticipata di una quantità P_i
  - Infine, nel caso la cpu sia in idle, tutti i server passano in stato I.

Il modo di incrementare la variabile V_i offre un'intuitiva chiave di lettura dell'algoritmo: se U è pari a 1 tutti i server sono attivi e quindi il singolo server S_i potrà eseguire $U_i * P_i$ unità di tempo ogni periodo P_i; viceversa, se il sistema è sottoutilizzato ($U < 1$), si possono attivare i meccanismi di recupero della larghezza di banda e il server ora può eseguire per $U_i * P_i/U$, sempre nello stesso periodo. La V_i va a misurare queste differenze, crescendo più lentamente quanto maggiore è il sotto-utilizzo del sistema, e quindi permettendo al server di rimanere in esecuzione per un lasso di tempo più lungo.

Questo meccanismo però rimane valido solo se V_i viene aggiornata ad ogni cambiamaneto di stato dei server presenti nel sistema, dato che questi influenzano il valore di U e di conseguenza il calcolo della V_i; in mancanza di questo accorgimento, potrebbero essere erroneamente recuperata parte della banda futuradi un server inattivo.

Rispetto al CBS di default, l'algoritmo GRUB tende ad eseguire mediamente un job per intervalli più lunghi, migliorandone mediamente i tempi di completamento.

3.9.2 Algoritmi semplificati di recupero

Nonostante i precedenti algoritmi rappresentino soluzioni valide per il recupero delle risorse, il loro costo di implementazione in termini di risorse

è elevato, e questo li rende di difficoltoso impiego in dispositivi embedded aventi limiti particolarmente stringenti in fatto di risorse disponibili.

In questi frangenti possono risultare più convenienti algoritmi computazionamente semplificati, come i due che andiamo a presentare, entrambi elaborati sempre tramite modifiche a server di tipo CBS.

Advancing server deadline

Un problema che affligge i server CBS si presenta nel caso di frequenti posticipi della propria deadline, il cui effetto è di abbassare notevolmente la priorità del server; questo avviene generalmente quando il server tenta di usare solamente il proprio budget, senza sfruttare anche quello di altri server sotto-utilizzati, e un esempio lo possiamo vedere in Figura 3.5: il primo server S_1 riceve cinque richieste di esecuzione ravvicinate e, poichè al secondo è ancora in idle, le primie 4 sono servite grazie a tre successivi posticipi della deadline, che alla fine risulta pari a $d = 16$. Arrivando in $t = 4$ il primo task servito da S_2, la quinta e ultima richiesta deve essere fortemente posticipata; si noti inoltre che il secondo server non viene assolutamente sfruttato nell'intervallo $[0, 4]$.

L'idea alla base della soluzione proposta si fonda sul seguente lemma: data una qualsiasi schedulazione σ, se si ha un intervallo $[t_0, t]$ (anche nullo) in cui il sistema è in stato idle, la schedulabilità dei job con istante di release maggiore o uguale a t non è compromessa dai job schedulati prima di t; in questo modo, t può essere designato come un nuovo inizio per l'analisi della schedulabilità, ignorando tranquillamente quanto accaduto prima.

Nell'esempio visto, questo consente di far ripartire agli istanti $t = 1, 2, 3, 4$, che precedono l'arrivo del primo task per S_2, il server S_1: di conseguenza, vengono eliminate le posticipazioni delle deadline e viene migliorato il tempo di risposta all'ultimo dei 5 task richiesti, che passa da $t = 13$ a $t = 4$, come illustrato in Figura 3.6

Budget adjustment

Il metodo riprende l'idea alla base dell'algoritmo CASH ma senza richiedere il mantenimento di una coda per le capacità residue; questa viene eliminata osservando l'elevata probabilità che sia solo il successivo server CBS attivo con la priorità più elevata a fare uso di eventuali capacità residue ereditate dal server precedente.

Al comportamento di base di un CBS, in base alla precedente osservazione, viene aggiunta una sola ulteriore regola: se il server corrente S_i passa in stato idle con ancora una capacità residua c_r, questa sarà resa disponibile ad un eventuale server S_j presente nella coda dei server pronti gestita con protocollo EDF; in caso S_j non esista, la capacità residua non viene persa.

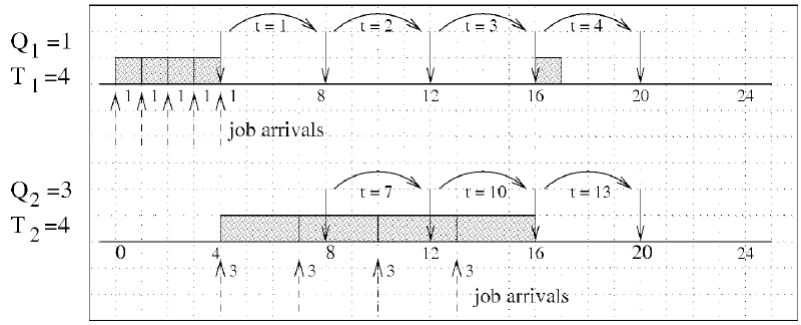


Figura 3.5: Situazione di forte posticipo della deadline per il primo server

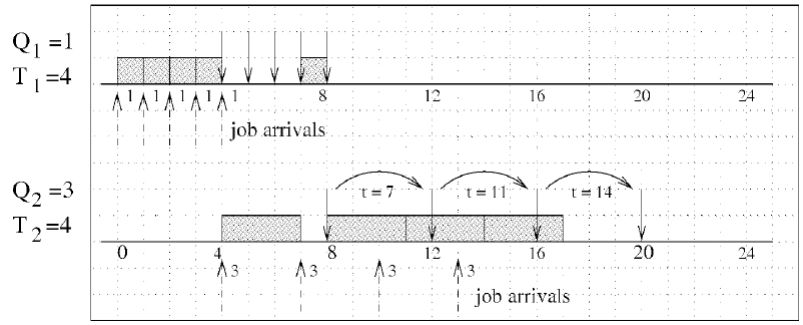


Figura 3.6: Impatto del server CBS con advancing deadline

Tale traferimento è sempre possibile in quanto l'algoritmo EDF ordina i server per deadline crescenti, e quindi la capacità residua di S_i è abbinata ad una deadline inferiore rispetto a quella di $S - j$.

La principale differenza rispetto al CASH, oltre al riutilizzo di al più una sola capacità residua, consta nell'impossibilità di attuare un tale riutilizzo se esiste un intervallo non nullo nel quale il sistema è in stato idle, proprio a causa della mancanza di una struttura di memorizzazione.

3.10 Qualità del servizio

Per fornire una definizione accettabile di qualità del servizio (QoS) è necessario innanzitutto trovare un modello che permetta di mappare univocamente gli aspetti soggettivi legati alla qualità percepita con valori numerici oggettivi.

3.10.1 Il modello Q-RAM

Il modello Q-RAM (*Q*oS based resource allocation model) costituisce un framework generale che permette di descrivere molteplici aspetti della qualità di servizio, mapparli in una serie di valori numerici (utilità) e utilizzare tali valori per determinare, se possibile, la corretta distribuzione delle risorse di sistema per ottenere il lvello qualitativo desiderato.

Due sono gli aspetti principali che caratterizzano il Q-RAM:

- permette di definire applicazioni che devono soddisfare a più tipologie di parametri. Questo segna un punto di svolta rispetto al modello real-time, in cui tutto viene determinato in base alla sola priorità. Un esempio di applicazione può essere quella di un riproduttore di file audio, che oltre a dover rispettare vincoli di tempo (come mantenere sempre un livello di dati pronti per essere riprodotti), può fornire diverse frequenze di campionamento, influenzando quindi la qualità dell'audio prodotto. Generalmente a questi parametri possono venire associati dei livelli minimi e massimi all'interno dei quali l'applicazione deve sempre mantenersi.

- Considera che ogni applicazione può richiedere diversi tipi e quantità di risorse per la sua esecuzione, e che spesso diminuire l'utilizzo di una implica un maggiore bisogno di un'altra; basti pensare alla trasmissione dati, nella quale i metodi di compressione comportano una minor utilizzazione della larghezza di banda a fronte di un maggiore impegno per la decodifica dei dati da parte della cpu.

In modo più formale, Q-RAM modella un sistema come un insieme di applicazioni $\{\tau_!, \tau_2, ..., \tau_n, \}$ (considerate costituite da un solo task) ed un insieme di risorse $\{R_1, R_2, ..., R_m\}$, dove ogni risorsa R_i può essere o meno

condivisa tra i task e ha un capacità massima R_i^max (definita dalle specifiche di sistema).

Obiettivo dell'algoritmo di allocazione delle risorse è trovare una distribuzione ottimale in funzione di un qualche criterio prestabilito; l'allocazione risultante sarà quindi una matrice $R_{i,j}$ i cui elementi indicano la porzione della risorsa R_j assegnata al task τ_i, tale per cui $\forall j, \sum_{i=1}^{n} R_{i,j} \leq R_j^max$.

Il criterio di ottimalità viene poi definito applicando a ciascun task τ_i una funzione di utilità $v_I(R)$ dipendente dalla particolare distribuzione di risorse $\{R_{i,1}, R_{i,2}, ..., R_{i,m}\}$ ad esso associata e non decrescente in funzione di ciascuna di esse (se una risorsa aumenta, la funzione non può assumere valori decrescenti). L'utilità globale di sistema si ottiene poi come $v_i = \sum_{i=1}^{n} w_i * v_i(R)$, dove w_i è pari all'importanza dell'applicazione τ_i.

Il problema di trovare una distribuzione può essere quindi definito come quello di trovare una matrice per cui la somma delle porzioni di ciascuna risorsa non superi la disponibilità totale della stessa, ad ogni applicazione sia garantita la minima quantità di ogni risorsa richiesta e la funzione di utilità sia massimizzata.

La soluzione si rivela complessa nel caso generale, ma può essere resa più trattabile con vincoli aggiuntivi. Nel caso di sistemi hard real-time dove le risorse sono allocate in modo statico si possono trovare soluzioni off-line e quindi la complessità del calcolo passa in secondo piano rispetto all'esigenza di trovare una soluzione il più prossima a quella ottimale.

La trattazione si pone ad un maggiore livello di difficoltà nel caso di risorse gestite in modo dinamico, come illustreremo nel prossimo paragrafo.

3.10.2 QoS e gestione dinamica delle risorse

Applicata nel caso di sistemi la cui natura è intrinsecamente dinamica (come nel caso del soft real-time), impone che il problema di ottimizzazione della distribuzione delle risorse sia risolto a runtime da un task specifico detto *QoS manager*.

Quest'ultimo si compone di due parti distinte:

1. **Meccanismo**: usato per assegnare una porzione specifica di risorse a ciascun task basandosi su una delle due possibili tecniche:
   - modificando i parametri dello schedulatore: questo consente di non modificare l'applicazione, ma lega in modo indissolubile il QoS manager al particolare algoritmo di scheduling adottato dal sistema.
   - adattando le applicazioni al QoS desiderato: così facendo si conserva l'indipendenza dallo schedulatore ma ogni applicazione deve essere in grado di adattarsi a diversi livelli di qualità del servizio e passare dinamicamente da uno all'altro a seconda delle mutate richieste del gestore.

2. **Politica**: definisce le modalità con cui il QoS manager partiziona le risorse tra le applicazioni; essa può essere determinata risolvendo on-line un definito problema di ottimizzazione o appoggiandosi a qualche sorta di euristica, la cui elaborazione può risultare complessa e richiedere quindi tempi o risorse computazionali elevate relativamente a quelle presenti nel sistema.

In generale, il QoS manager può basare le proprie decisioni sulla distribuzione delle risorse secondo due differenti modalità:

- riferendosi alle richieste esplicite dichiarate dalle applicazione, nel qual caso sia ha la necessità che ogni applicazione sia in gardo di determinare questi parametri nel momento in cui viene portata ini esecuzione e quindi il gestore prenderà le proprie decisione ogni qualvolta un processo entrerà nel sistema o ne uscirà, cambierà il livello di servizio richiesto, i propri requisiti o i livelli di risorse consumate.

- facendo perno su meccanismi di feedback e quindi monitorando periodicamente le prestazioni del sistema e il consumo di risorse per variare dinamicamente i parametri della funzione di utilità adottata.

3.10.3 Smooth rate adaptation

Nelle applicazioni soft real-time che coinvolgono pesantemente l'interazione umana, come la fruizione di flussi multimediali, la qualità del servizio non è determinata solamente dal livello assoluto di prestazioni, ma anche dal modo in cui questo varia in funzione del carico di sistema; nei casi considerati risulta sempre preferibile mantenere un passaggio graduale tra valori prestazionali diversi.

Ad esempio, in attività periodiche il livello di qualità del servizio può essere determinato dalla frequenza di ciascun processo, e ltali modifiche possono essere rese graduali apportandole attraverso vari passaggi parziali intermedi.

Tralasciando i dettagli del modello (ritrovabili alle pagine 207-210 di [2]), derivato da una ulteriore generalizzazione dei modello elastico proposto da Buttazzo e Abeni, ci limiteremo a vederne l'implementazione e alcune proprietà estrapolate dai risultati di alcuni test.

La realizzazione si compone di un task che si occupa di raalizzare il modello elastico (*Elastic Manager* (EM)), che si occupa di variare in modo immediato il periodo dei task che ne richiedono la modifica, e di un secondo processo (*Damper Manager*, (DM)) con il quale questa transizione viene resa graduale secondo una funzione impostata dall'utente.

Il DM è inizialmente in stato idle e viene attivato solo quando vengono richieste le modifiche di periodo da parte dei task; una volta che l'EM ha determinato la nuova frequenza, il DM viene eseguito con periodo T_{DM} e realizza la transizione dal vecchio al nuovo periodo in un intervallo di transizione pari a $N * T_{DM}$, dove N è un parametro impostabile al momento della

chiamata. Ad ogni step, il DM richiama comunque l'EM per assicurarsi di mantenere una schedulazione fattibile.

I test condotti sul comportamento di un tale meccanismo, incentrati specialmente sull'utilizzo di diverse leggi matematiche che sottendono a varie tipologie di transizione, evidenziano che:

- una transizione di tipo lineare permette di ottenere una più graduale variazione dei periodi dei task quando un processo già attivo richiede la modifica della propria frequenza di attivazione.

- viceversa, una transizione basata su legge esponenziale usata per ridurre i periodi di tutti i task attivi del sistema quando un nuovo processo viene eseguito per la prima volta, risulta più efficace e riduce il ritardo di completa attivazione di quest'ultimo.

3.11 Scheduling con feedback(adattabilità)

Come anticipato nel paragrafo precedente, un modo per gestire l'imprevedibilità di un sistema è utilizzare un meccansmo di feedback al fine di adattare dinamicamente il comportamento dello schedulatore in funzione di una qaulche metrica QoS.

Per rendere operativo questo approccio serve definire un algoritmo di scheduling, un indice di qualità del servizio e i parametri dello scheduler su cui agire per variarne il comportamento.

Per illustrarne vantaggi e problematiche connesse, ne vedremo brevemente alcune implementazioni.

3.11.1 FC-EDF

L'EDF con *Feedback control*, introdotto inizialmente per gestire sistemi con risorse scarse rispetto al carico elaborativo da sopportare, utilizza come parametro QoS la frequenza delle violazioni delle deadline $M(t)$, ovvero il rapporto tra numero di deadline violate e totali considerate in una data finestra temporale di dimensioni fissate.

I task presenti nel sistema devono dichiarare due o più livelli di QoS, distinti per valori del tempo di esecuzione o di qualità dei risultati forniti; lo scheduler, in base al valore di $M(t)$, decide se rifiutare o meno l'esecuzione di nuovi task e, nel caso questo venga accettato, ne determina anche il livello di qualità del servizio con cui eseguirlo.

L'equazione [10] alla base del calcolo della variazione ΔU di utilizzazione della cpu è pari a:

$$\Delta U = C_p * error(t) + C_i * \sum_{IW} error(t) + C_D * \frac{error(t) - error(t - DW)}{DW}$$

dove $error(t) = M_s - M(t)$ con M_s violazioni desiderate, IW è la finestra temporale usata per la sommatoria delle deadline in essa registrate e DW quella impiegata per valutare la variazione dello scostamento dai valori desiderati di violazioni ammesse.

I tre addendi della formula considerano rispettivamente cambiamenti proporzionali allo scostamento totale, allo scostamento considerato nell'ultimo periodo IW e alla variazione dell'errore nell'arco temporale DW.

Le valutazioni sperimentali dimostrano che l'algoritmo riesce a contenere in modo soddisfacente la frequenza delle deadline violate anche in presenza di variazioni forti del carico di sistema, mantenendone anche una utilizzazione elevata. Per contro, in condizioni di frequenza nulla delle deadline violate, non riesce a distinguere se un sistema è o meno scarsamente utilizzato, propro perchè controlla solamente il numero di deadline violate e non il valore della $U(t)$ globale; l'algoritmo, in quesete condizioni, prova ad aumentare l'utilizzazione, tornando ciclicamente a presentare nuove violazioni delle deadline (*limit cycle*).

Un suo successivo sviluppo, l'$FC{-}EDF^2$, colma questa lacuna inserendo un secondo sistema di controllo che calcola la variazione di utilizzazione ΔU in base all'utilizzazione stessa; tra le due ΔU ricavate, viene ovviamente selezionata quella di entità minore, in modo da raggiungere un comportamento stabile.

3.11.2 Adaptive reservation

Algoritmi come il precedente sono limitati dal fatto di controllare un parametro QoS globale per il sistema, senza entrare nel merito delle prestazioni delle singole applicazioni presenti.

Per ovviare a questa mancanza possiamo ricorrere all'utilizzo dei server CBS3.6.2, i quali permettono di controllare in modo indipedente i processi del sistema, Per contro, i parametri di ciascun server sono fortemente legati a quelli temporali del task e non vi è modo, per chi progetta il sistema, di esprimere altre relazioni globali tra i task stessi (come ad esempio la loro importanza all'interno del sistema).

L'*Adaptive reservation* riesce a coniugare le due necessità assegnando a ciascun task τ_i un server CBS $(U^S_{i,j}, P^S_i)$ e, al termine di ogni job $\tau_{i,j}$ del task osserva un certo valore $\epsilon_{i,j}$ e calcola $U^S_{i,j+1}$ in funzione di $(U^S_{i,j}, \epsilon_{i,j})$ ed eventualmente altri parametri.

Il valore $\epsilon_{i,j}$ è detto *scheduling error* ed è pari alla differenza tra la deadline assegnata al job dal server, $d^S_{i,j}$, e la deadline soft del job, $r_{i,j+T_i}$; ricordiamo che, se l'insieme dei server è schedulabile, viene garantito il rispetto delle deadline assegnate ma non di quelle soft specifiche di ogni job.

Il valore della grandezza $\epsilon_{i,j}$ offre una misura della bontà del servizio dato da ciascun server al task servito: se $\epsilon_{i,j} = 0$ tutti i job sono completati entro le proprie deadline, mentre un eventuale valore positivo indicherebbe la pre-

senza di violazioni e quindi di una banda insufficiente messa a disposizione; controllando $\epsilon_{i,j}$ si possono quindi operare gli opportuni aggiustamenti alla banda $U_{i,j}^S$ offerta a ciascun job.

D'altro canto, deve essere presente un ulteriore meccanismo globale in grado di controllare che la sommatoria delle utilizzazioni così determinate sia non superiore alla soglia di utiiizzazione relativa all'algoritmo di schedulazione globale (1 nel caso di EDF). Se così non fosse, deve essere attivata una tecnica di compressione che, in base all'importanza w_i attribuita al task, ricalcoli le bande assegnate in modo da rendere fattibile l'insieme dei server attivi secondo una qualche funzione da definire. Così facendo, in caso di overload, questo tende ad essere confinato nei task di importanza minima, visto che questa determina anche il livello di compressione usato (maggiore in caso di peso minore).

3.11.3 Adattamento a livello di applicazione

Gli algoritmi con feeedback visti nelle due sezioni precedenti agiscono sempre modificando il comportamento dello schedulatore. Questo non permette però di scegliere in modo esplicito la strategia migliore con cui regolare il carico computazionale del sistema: può capitare infatti che ad un task venga assegnata una banda insufficiente per soddisfare i parametri QoS richiesti.

Un tale situazione può essere risolta solo aumentando la banda U_i per soddisfare i requisiti QoS (come nel caso dell'*Adaptive reservation*) o diminuendo il tempo cpu richiesto dal task in modo da adattarlo alla banda assegnata (come in FC-EDF). La modifica introdotta dal nuovo algoritmo mira a far determinare esplicitamente ad ogni applicazione l'entità del riscalamento delle proprie richieste computazionali per rimuovere l'overload e mantenere la schedulazione calcolata fattibile, in maniera tale che lo scheduler globale non se ne occupi.

In questo modo, in situazioni di overload, ciascun task può abbassare il livello dei propri requisiti in modo controllato, eliminando al contempo il sovraccarico di sistema.

L'*Adaptive reservation* e l'adattamento a livello di applicazione, lavorando su ambiti diversi (globale il primo, locale al task il secondo) sono stati integrati assieme in un approccio chiamto *hierarchical adaptation*: questo presenta i vantaggi di entrambi, consentendo ad una applicazione di scalare le proprie richieste nel caso lo scheduling globale non gli riservi banda sufficiente.

Si presenta però un nuovo problema, illustrato nell'esempio seguente: ipotizzata la presenza dei task τ_1 e τ_2, in caso di overload lo scheduler globale può decidere di abbassare U_1, alla quale il task τ_1 reagisce localmente abbassando i propri parametri QoS. Al termine dell'overload, però, lo scheduler globale può intervenire aumentando U_2, con il risultato che alla fine il secondo task sottragga banda al primo, il quale, malgrado la situazione

di overload sia terminata, non ha più la possibilità di ritornare al livello di qualità del servizio precedente al sovraccarico.

Una soluzione parziale consiste nel far intervenire il meccanismo locale in modo ritardato, così da essere applicato solo per overload prolungti nel tempo.

3.11.4 Stimatori di carico

In ambienti fortemente dinamici, molti degli algoritmi visti potrebbero risentire dell'impossibilità di determianre precisamente il tempo di esecuzione (medio o massimo) dei vari task; si pensi ad esempio al modello elastico che fa un forte uso del WCET.

In questi frangenti una soluzione possibile è aggiungere al sistema un processo specifico dedicato a ricavare una stima on-line di questi valori, impiegando funzioni offerte dal sistema operativo per misurare i tempi di esecuzione dei processi. La stima ricavata può essere successivamente utilizzata come parametro di osservazione su cui costruire un meccanismo di adattamento di carico.

Ad esempio, un task τ_i potrebbe essere creato inizialmente alla minima frequenza (teoricamente nulla) e, alla fine di ogni periodo (ovvero dopo ciascun job del task), si determinerebbero i tempi medi c_i e massimi C_i di esecuzione; con i valori ricavati si potrebbe stimare il budget Q_i da assegnare al server che si occupa del task e ricavare l'utilizzazione globale $U = \sum_i \frac{Q_i}{P_i}$, da impiegare poi in algoritmi come il modello elastico per riadattare la frequenza del task.

L'efficienza dell'intero algoritmo si basa sull'efficacia dello stimatore del tempo di esecuzione, la cui costruzione può beneficiare delle seguenti osservazioni:

- se la C_i stimata fosse prossima alla WCET avrei una bassa frequenza di deadline violate ma contemporaneamente una sotto utiizzazione del sistema.
- se invece fosse vicina a c_i, avremo esattamente la situazione opposta.

In definitiva, viene generalmente utilizzata l'equazione $Q_i = c_i + k * (C_i - c_i)$ con $k \in [0, 1]$, che tramite il parametro k permette di bilanciare il comportamento tra deadline violate e throughput.

L'approccio è ottimale in caso di task con job dai tempi di esecuzione fortemente variabili, mentre lo è meno se questi sono costanti; infatti, c_i impiegherà un certo tempo per raggiungere un valore prossimo a C_i e quindi il task potrà manifestare violazioni di deadline anche frequenti nel transitorio iniziale. In questa situazione un valore di k pari a 1 assicura un miglior comportamento del task.

Capitolo 4

Il kernel Linux

4.1 Sommario

In questo capitolo verranno illustrate la struttura e le componenti principali del kernel Linux, riservando un'attenzione particolare a quelle che ne influenzano la maggiore o minore propensione ad un adattamento ai sistemi soft real-time; dove non esplicitamente dichiarato, si farà riferimento al libro di Love Robert [4].

Gli eventuali riferimenti al codice sono relativi alla versione stabile 3.7.6 dei sorgenti del kernel, l'ultima disponibile al momento della redazione della tesi.

4.2 Storia di Linux

Prima di addentrarci nell'analisi del kernel Linux, risulta utile fornire una descrizione generale di Unix, un sistema operativo nato oramai più di 40 anni fa e con il quale lo stesso kernel Linux condivide molte caratteristiche strutturali. La longevità e la flessibilità di Unix sono attribuibili alle seguenti scelte progettuali applicate nella sua realizzazione:

- Presenta di un numero limitato di chiamate a sistema.
- La struttura cardine del sistema è il file, utilizzato per rappresentare qualsiasi elemento, sia esso un dato o un dispositivo fisico, Questa scelta consente di avere un meccanismo semplice e omogeneo per la manipolazione ogni componente del sistema.
- È realizzato per la gran parte in linguaggio C e questo ne ha notevolmente incrementato il grado di portabilità su architetture differenti.
- La creazione di nuovi processi è un'operazione rapida e gestita tramite una interfaccia costituita da una sola chiamata a sistema, la fork().

- Integra un sistema semplice e robusto per la gestione delle comunicazioni tra processi.

Col il tempo Unix si è inoltre evoluto andando ad integrare funzionalità quali il multithreading, il multitasking con prelazione, la paginazione e la virtualizzazione della memoria, il supporto a librerie consivise e al protocollo di rete TCP/IP; in questo modo il sistema operativo stesso ha saputo mantenere un grado di flessibilità tale da permetterne la diffusione sia su server con migliaia di cpu che su piccoli sistemi embedded.

La nascita del kernel Linux può essere collocata nel 1991 quando uno studente finlandese, Linus Torvald, dà il via al suo sviluppo con l'obiettivo si sopperire ad alcune caratteristiche da lui ritenute poco soddisfacenti del sistema operativo Minix utilizzato in ambito didattico; quest'ultimo, oltre a presentare soluzioni strutturali non condivise da Torvalds, era legato ad una licenza che impediva di modicarne il codice e ridistribuire liberamente ad altri utenti tali variazioni.

Sul finire del 1991 Torvalds rende disponibile una prima versione del suo lavoro riuscendo in breve tempo a trasformarlo in un progetto di gruppo grazie al coinvolgimento di un buon numero di sviluppatori.

Con Unix, il kernel Linux condivide buona parte delle strutture che lo compongono e la presenza su una grande varietà di calcolatori, pur includendo delle peculiarità distintive, la maggiore delle quali riguarda l'essere un progetto libero distribuito con licenza GPL v2 che ne garantisce la libera fruibilità del codice sorgente e la possibilità di apportarne modifiche con il vincolo però di rilasciarle sempre sotto la medesima licenza. Questo, se da una parte consente un facile accesso al codice e quindi la possibilità di analizzarlo in modo approfondito, dall'altra limita una collaborazione diretta di aziende che intendono mantenere la proprietà intellettuale delle particolari funzionalità integrate.

4.3 Concetti base

Quando parliamo di sistema operativo andiamo ad indicare quella parte del software dedicata a fornire gli strumenti per l'amministrazione e l'utilizzo di base dell'intero sistema e quindi comprendente elementi quali il kernel, i driver dei dispositivi, un sistema di boot, un terminale per l'interazione con l'utente e utilità per la manipolazione dei file.

Il kernel (o nucleo del sistema) è quella parte di software che si occupa di rendere disponibile tutta una serie di servizi mirati all'amministrazione e alla gestione del funzionamento dell'intero sistema.

Nello specifico, come illustrato anche dalla Figura 4.1, possiamo delineare tre compiti principali che il kernel deve soddisfare:

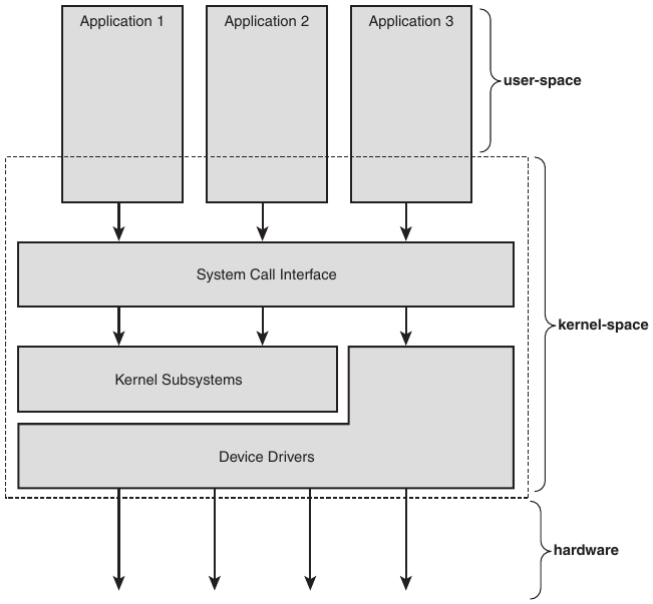


Figura 4.1: Struttura generale del kernel

- Fornire un'interfaccia comune con la quale le applicazioni interagiscono con il resto del sistema attraverso un insieme di funzioni indicate come *chiamate di sistema* (*system calls*).
- Consentire l'accesso ad un'ampia varietà di dispositivi e alle loro peculiari funzionalità attraverso l'integrazione di driver specifici e alla disponibilità di meccanismi di comunicazione fra questi e il sistema operativo.
- Regolare, attraverso una serie di servizi e sottosistemi, le modalità di esecuzione delle applicazioni, le comunicazioni inter-processo e la distribuzione delle risorse di sistema necessarie al completamento delle stesse.

4.3.1 Tipologie di kernel

I vari kernel esistenti possono presentare strutture interne molto diversificate, raggruppabili però in alcuni principali sottogruppi:

1. **kernel monolitici**: costituiti da un singolo processo eseguito in un unico spazio di memoria. Tutti i servizi esistono e sono eseguiti all'interno di questo ambiente e le comunicazioni tra di essi risultano quindi semplici e dotate di buone prestazioni velocistiche.
2. **microkernel**: vedono le varie funzionalità del kernel suddivise in processi specifici detti server; di questi, solo quelli che necessitano stret-

tamente di permessi elevati sono eseguiti in modalità kernel, mentre tutti gli altri sono mantenuti in modalità utente; questo comporta l'impossibilità di comunicazioni dirette e la conseguente necessità di implementare un meccanismo separato di intercomunicazione.

Questo però va ad introdurre un elevato overhead (determinato in larga misura dai frequenti cambi di contesto) e incide negativamente sulle prestazioni. D'altro canto, è possibile caricare in memoria solo i server necessari alle richieste, occupandone quindi solo la quantità strettamente necessaria; inoltre, eventuali malfunzionamenti di un singolo server hanno ricadute limitate all'esecuzione del server stesso e non provocano il blocco dell'intero sistema come nel caso dei kernel monolitici.

3. **modulari**: è il gruppo a cui appartiene il kernel Linux che, pur avendo una struttura monolitica, supporta la capacità di caricare o rilasciare in modo dinamico porzioni di codice dette moduli che consentono l'integrazione di nuove funzionalità non appena queste vengono richieste; esso quindi racchiude i vantaggi del kernel monolitico garantendo al contempo una migliore occupazione di memoria, al costo di un decremento prestazionale (generalmente irrisorio) dovuto al tempo di caricamento dei moduli.

4.3.2 I contesti di esecuzione

Quando si parla di kernel è necessario precisarne le differenze principali rispetto alle comuni applicazioni. Una prima distinzione si evidenzia in quelli che vengono definiti contesti di esecuzione.

I sistemi attuali, dotati di unità di protezione della memoria (MMU) permettono di rendere disponibile al kernel un ambiente indicato come kernel-space, composto da uno spazio di memoria privato e un completo accesso alle funzioni hardware.

Viceversa, le applicazioni utente sono eseguite in un ambiente, lo user-space, con limitazioni tali da costringerle ad eseguire operazioni sul sistema solamente tramite l'intermediazione del kernel, attraverso un meccamismo prestabilito di comunicazione che si esplica nell'insieme delle *chiamate di sistema*; tale situazione, nella quale il kernel opera per conto di un processo, si definisce tecnicamente *contesto di processo* o *process context*. L'intervento delle chiamate di sistema implica sempre un passaggio tra la modalità utente e la modalità kernel, chiamato *mode switch* le cui modalità e tempistiche di esecuzione dipendono dalla particolare architettura della cpu.

Questa scelta privilegia una maggiore stabilità di sistema poichè tutte le operazioni sono controllate dal kernel; d'altro canto, il fatto di poter eseguire certe operazioni privilegiate solo tramite un intermediario e non in modo

diretto può comportare per le applicazioni utente un possibile peggioramento dei tempi di esecuzione o delle latenze.

Il kernel, oltre alle applicazioni, deve gestire anche l'hardware del sistema; questo compito viene svolto attraverso l'impiego di due meccanismi: i driver di dispositivo, contenenti il codice che abilita le specifiche funzionalità della data periferica, e un particolare meccanismo di comunicazione basato sulle interruzioni.

Queste ultime sono costituite da un insieme di segnali (distinguibili tramite identificativi numerici) tramite i quali i dispositivi hardware possono avvisare del verificarsi di un determinato evento e ai quali il sistema reagisce interrompendo l'elaborazione corrente del processore e andando ad eseguire opportune routine del kernel (interrupt handler) dedicate alla gestione di tali segnali.

In definitiva, la presenza di diverse modalità di esecuzione comporta che, in ogni istante, un processore possa trovarsi in una sola delle seguenti condizioni:

1. Eseguire codice utente in spazio utente.
2. Eseguire una chiamata di sistema per conto di un qualche processo, e quindi essere in modalità kernel in un contesto di processo.
3. Eseguire un gestore di interruzioni, e quindi trovarsi in kernel mode ma in un contesto di interruzione.

Potrebbero sorgere dubbi rispetto al comportamento nel caso non vi sia alcun processo pronto per essere eseguito; in questo caso, l kernel mette a disposizione un task particolare (*processo idle*) che viene eseguito in queste particolari condizioni, portando così il sistema ad operare nella seconda modalità elencata.

4.3.3 Kernel e applicazioni utente

Oltre alla modalità di esecuzione precedentemente vista, esistono altre differenze tra il kernel e le applicazioni utente, che possiamo così riassumere:

- Il kernel non ha accesso alle librerie C ne agli headers standard: le librerie C complete risultano troppo grandi e poco prestanti per il kernel. Molte delle funzioni di queste librerie sono implementate anche all'interno del kernel (vedere gli header nelle cartelle /include e /arch/<architettura>/include). Altre sono implementate con caratteristiche differenti e con nomi cambiati (vedi `printf()` e `printk()`)
- È implementato in linguaggio GNU C e non in C puro: non si usa l'ANSI C, ma vengono sfruttate tutte le estensioni del linguaggio incluse nel compilatore gcc (funzioni inline, inline assembly attraverso la

direttiva asm(), le direttive `likely()` e `unlikely()` che ottimizzano le condizioni degli if nei casi rispettivi che una condizione abbia maggiore o minore probabilità di risultare vera). La sua portabilità è un parametro rilevante: questo lo si ottiene assicurandosi che il codice C indipendente dall'architettura risulti compilabile su ciascuna piattaforma mentre quello specifico vada opportunamente confinato in cartelle specifiche.

- Non può usufruire della protezione della memoria: se il kernel compie un accesso non permesso ad un'area di memoria risulta un errore di oops, uno dei bug più comuni. Inoltre, la memoria del kernel non è sottoposta a paginazione e quindi ogni bit usato è un bit in meno a disposizione.

- Non è in grado di eseguire calcoli in virgola mobile (o il processo è complesso): ciò richiederebbe il salvataggio e il ripristino manuale dei registri dedicati ai calcoli in virgola mobile (per i processi utente è il kernel che gestisce la transizione da calcola intero a calcolo in virgola mobile).

- Il kernel ha uno stack di dimensione piccola e fissa per ciascun processo: mentre i processi godono di uno stack grande e dalle dimensioni incrementabili, il kernel può utilizzare un solo stack (per ciascun processo in modalità kernel) di dimensioni ridotte, dipendenti dall'architettura (2 pagine di 4 o 8 KB ad esempio su cpu x86, rispettivamente a 32 e 64 bit)

- Poichè supporta prelazione, interruzioni asincrone e sistemi SMP la concorrenza e la sincronizzazione sono due elementi fondamentali da considerare nella realizzazione delle sue componenti. Il kernel infatti permette l'accesso concorrente alle risorse e quindi va opportunamente gestita la sincronizzazione degli accessi per evitare fenomeni quali le race condition. In particolare vanno gestite le seguenti caratteristiche:

  - La prelazione implica la continua rischedulazione dei processi, che devono quindi essere sincronizzati rispetto a tale meccanismo; questo vale anche per processi del kernel stesso, poichè anch'essi sono soggetti a prelazione.
  - Il supporto all'SMP implica impedire a più cpu o core di accedere simultaneamente alla stessa risorsa. Le interruzioni sono però asincrone rispetto alla cpu e l'esecuzione del relativo gestore potrebbe andare ad utilizzare risorse detenute dal processo appena interrotto.

4.3.4 Linux e il soft real-time

Linux, come abbiamo visto, nasce come kernel indirizzato a sistemi general-purpose e quindi privo di strutture per la gestione di ambienti real-time.

Per valutarne la capacità di adattarsi comunque al supporto del soft real-time, si fa riferimento [5]al *kernel response time*, ossia all'intervallo di tempo compreso tra la generazione di una interruzione(sezione 4.7.1) e l'istante in cui il task che la attendeva riprende la sua esecuzione. Possiamo pensare ad esempio ad un processo in attesa del termine di una operazione di I/O su disco; l'interazione fra questa e il processo stesso si esplica in quattro passaggi successivi:

- l'operazione viene completata e di conseguenza il dispositivo solleva una interruzione.
- l'ISR (sezione 4.7.1) determina il dispositivo che ha generato l'interruzione e richiama la specifica funzione per gestirlo; questa a sua volta va a risvegliare il task in attesa e lo pone nella coda di esecuzione dello schedulatore (sezione 4.5)
- il kernel richiama la funzione `schedule()` non appena raggiunge un punto nel quale è ammessa la schedulazione.
- lo scheduler porta in esecuzione il task in attesa.

Dall'esempio risulta che il tempo di risposta del kernel è determinato da quattro distinte componenti:

1. **Latenza di interruzione**:intervallo fra la generazione dell'interruzione e l'istante in cui viene richiamata l'ISR.
2. **Durata dell'ISR**: tempo di esecuzione dell'ISR.
3. **Latenza dello scheduler**: è il tempo compreso tra l'istante in cui termina l'ISR e quello in cui la funzione di schedulazione viene
4. **Durata dello scheduler**:l'ammontare del tempo impiegato per decidere il prossimo processo da portare in esecuzione.

La suddivisione appena vista vale naturalmente se il prossimo processo ad essere scelto sarà effettivamente il task in attesa dell'interruzione, altrimenti dovremmo conteggiare gli ulteriori ritardi dovuti alla presenza di processi con maggiore priorità di esecuzione.

La successiva analisi delle varie componenti del kernel verificherà come queste possano influire, positivamente o meno, sull'entità del tempo di risposta del kernel.

4.4 Processi e thread

Con il termine processo (o task) non si indica il solo codice eseguibile di una applicazione, ma il programma nell'istante in cui viene effettivamente eseguito dal sistema; esso quindi è composto da numerosi altri elementi, quali i file aperti ad esso associati, lo stato del processore, un proprio spazio di indirizzamento della memoria, uno o più thread di esecuzione e riferimenti a dati interni al nucleo del sistema.

I thread a loro volta costituiscono l'elemento portante del processo stesso: ogni thread possiede un proprio stack e un insieme di registri di esecuzione e può essere elaborato indipendentemente dagli altri; lo stesso schedulatore (sezione 4.5) manipola thread e non direttamente i processi. I thread permettono di avere più flussi di esecuzione in uno spazio di memoria condiviso, consentendo la programmazione concorrente e, nei sistemi multicore o multicpu, un reale parallelismo.

Tra di essi si distinguono i *kernel thread*, utilizzati dal kernel per realizzare operazioni periodiche da eseguire in background. Essi possono operare solamente in kernel-mode, vengono generalmente creati al boot del sistema e sono generati solamente da altri kernel thread; difatti il sistema li gestisce attraverso un unico processo detto *kthreadd*, definito in linux/kthread.h.

Il vantaggio della loro introduzione si esplica nel fatto di gestire processi periodici attivi durante l'intero arco di attività del sistema che richiedono permessi disponibili solo in modalità kernel, senza dover ricorrere a processi in user space (che necessiterebbero di operazioni di *mode switch* in concomitanza di ogni loro intervento [23]).

Sia i task che i thread non sono oggetti statici, ma presentano una loro evoluzione; il sistema deve essere quindi in grado di crearli, allocando le risorse necessarie, controllarne lo stato e provvedere al recupero delle risorse utilizzate una vlta terminati.

Altre due proprietà che il kernel Linux garantisce ai processi sono le seguenti:

- virtualizzazione del processore: il sistema offre al processo l'illusione di essere l'unico detentore del tempo di elaborazione cpu, anche se in realtà questo viene suddiviso tra tutti i task presenti.

- virtualizzazione della memoria: ogni processo può manipolare la memoria di sistema come se fosse completamente a sua unica disposizione.

I thread appartenenti allo stesso processo, invece, pur ricevendo ciascuno un proprio intervallo di tempo cpu, condividono l'utilizzo della medesima porzione di memoria.

Lo studio delle strutture usate dal kernel per rappresentare e gestire i processi e i thread può fornirci una prima indicazione di quante delle peculiarità relative all'ambito soft real-time siano effettivamente presenti o trasportabili

all'interno dei task di Linux; ne faremo quindi una rapida analisi riferendoci ad alcuni header del codice sorgente contenuti nella cartella /include.

4.4.1 Implementazione

L'elenco di tutti i processi presenti nel sistema viene mantenuta dal kernel in una lista doppiamente concatenata (task list), i cui elementi sono indicati con il nome di descrittori di processo, ovvero variabili particolari di tipo `task_struct` che consentono al sistema di rappresentare in modo completo lo stato di un task e di accedere ai vari parametri per manipolarli.

Ogni processo è identificato attraverso un valore numerico univoco detto *PID*, di tipo pid_t, con limite massimo di 32767 ma aumentabile fino a 4 milioni attraverso l'header linux/thread.h. Poichè i processi sono uno degli oggetti più frequentemente utilizzati, il kernel generalmente si riferisce ad essi non tramite il PID ma mantenendo un puntatore al descrittore associato (alcune architetture cpu ricche di registri come il PowerPC di IBM salvano questo puntatore su un registro dedicato per garantirne la massima velocità di accesso).

Inoltre, una delle operazioni più frequenti in un sistema operativo è la gestione del task corrente attraverso l'istanza della `task_struct` abbinata; per migliorarne l'efficienza, Linux mette a disposizione la macro current() la quale ne ritorna l'indirizzo. L'implementazione di tale macro viene lasciata dipendente dall'architettura, in modo da poter sfruttare eventuali meccanismi hardware: ad esempio, nelle cpu RISC ARM e PPC, dotate di molti registri interni, tale macro non fa altro che salvare l'indirizzo desiderato in un registro dedicato (rendendone quindi l'accesso istantaneo), mentre nelle architetture x86 l'indirizzo viene recuperato attraverso una struttura di tipo thread_info, definita in arch/<architettura>/include/thread_info.h, che viene posta sequenzialmente allo stack riservato dal kernel al processo e che contiene a sua volta un riferimento alla locazione di memoria del descrittore di processo. La sua introduzione è legata al meccanismo dello slab allocator (sezione 4.9.3) introdotto per risolvere problematiche di frammentazione della memoria.

Il campo *state* indica uno dei possibili stati, illustrati in Figura 4.2, nei quali un processo può venire a trovarsi durante l'evoluzione della sua esecuzione; tra di essi i principali sono:

- `TASK_RUNNING`: il processo è attualmente in esecuzione o nella coda ready (struttura in cui sono collocati i descrittori relativi ai processi che attendono solamente la disponibilità di tempo cpu per proseguire nella loro esecuzione).

- `TASK_INTERRUPTIBLE`: il processo è bloccato in attesa del verificarsi di una qualche condizione che gli permetta di ritornare allo stato di

running; è il kernel stesso che si occupa di verificare la manifestazione dell'evento e di riportare il processo in `TASK_RUNNING`.

- `TASK_UNINTERRUPTIBLE`: come il precedente eccettuato il fatto che il processo non verrà risvegliato nel caso di arrivo di un qualche segnale. Viene usato per attese brevi o che avvengono con interruzioni disabilitate.
- `__TASK_STOPPED`: il task viene fermato e non può più essere riportato in esecuzione.

L'elenco completo può essere trovato in /include/linux/sched.h

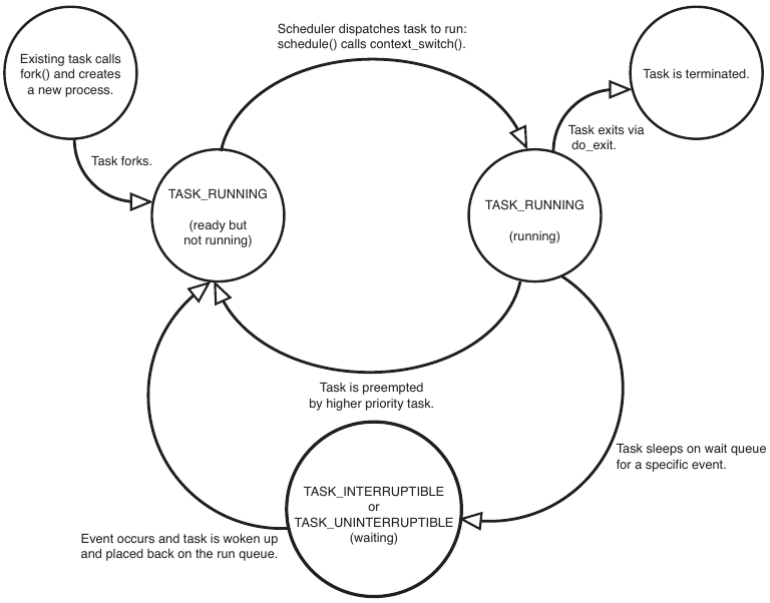


Figura 4.2: Relazioni fra gli stati dei task

4.4.2 Manipolazione di processi e thread

La creazione di un nuovo processo avviene in due fasi: la chiamata a `fork()` che crea una copia del processo padre (distinta dall'originale solo per il PID e alcune strutture non ereditate) e la chiamata `exec()`, che carica il codice del nuovo programma che si vuole lanciare. In questo modo tutti i processi sono legati tra loro in un albero gerarchico, il cui capostipite è rappresentato dal processo init, il primo ad essere creato ad ogni avvio di sistema.

L'implementazione della funzione fork si basa a sua volta sull'unica funzione `clone()`, alla quale vengono passati particolari parametri che specificano quali risorse i task padre e figlio debbano condividere e quali invece mantenere separate.

Questo ne consente di conseguenza il riutilizzo anche nel caso della creazione dei thread. I valori ammessi dai parametri della funzione„ di tipo bit, sono contenuti sempre nel file /include/linux/uapi/sched.h e ne riportiamo qui solo i più significativi:

- `CLONE_VM`: indica la condivisione dello spazio virtuale di memoria.
- `CLONE_FS`: condivisione delle info sul filesystem.
- `CLONE_FILES`: condivisione dei file aperti.
- `CLONE_SIGHAND`: condivisione dei gestori di segnali e dei segnali bloccati.
- `CLONE_PTRACE`: indica se continuare le operazioni di tracing anche sul processo figlio.
- `CLONE_VFORK`: indica che il padre vuole essere risvegliato dal figlio non appena questo rilascia la memoria condivisa.
- `CLONE_IO`: comporta la condivisione dello stesso contesto I/O, ovvero sono visti come un solo processo dallo schedulatore I/O del kernel ed il sistema permette loro di inframmezzare le richieste di input/output).

La fase di duplicazione nell'esecuzione di `fork()` è comunque dispendiosa poichè si deve allocare spazio e copiare la quasi totalità delle strutture del processo padre. In Linux si adotta la tecnica del *copy-on-write (COW)*, ovvero i due processi (padre e figlio) condividono le stesse strutture fino a quando uno dei due non necessita di modificarle; solo a questo punto sarà effettuata la copia della parte soggetta a modifica. In caso di assenza di modifiche, la chiamata `fork()` si limita a duplicare la tabella delle pagine del processo padre e creare un nuovo descrittore per il figlio, con un notevole risparmio di risorse di sistema e tempo di esecuzione.

Quando il processo termina, invece, il kernel deve provvedere a liberare tutte le risorse utilizzate dal processo nella sua esecuzione e non ancora rilasciate e comunicare lo stato di uscita del thread al processo padre. A questo punto, il processo è in stato zombie e occupa ancora lo stack kernel associato, la struttura `thread_info` e la `task_struct`, che verranno rilasciate non appena sarà confermata la ricezione dello stato di uscita da parte del padre. Nel caso il processo padre sia terminato per un qualsiasi motivo, il controllo dello stato di uscita viene affidato al processo init.

In ultima conclusione, nella gestione di processi e thread il kernel presenta i seguenti caratteri

- gestione unificata thread e task, che quindi riduce il numero di strutture da usare nel kernel e unifica l'interfaccia per le funzioni di controllo.
- accesso rapido al processo corrente grazie alla macro `current()`.

- alta flessibilità della funzione `clone()` che permette di specificare nel dettaglio quali componenti dei processi condividere e quali no.
- introduzione del meccanismo di copy-on-write: relativamente all'ambiente embedded soft real-time, esso assume un comportamento duale; se da una parte, infatti, consente un risparmio delle risorse annesse alla creazione di nuovi processi o thread, dall'altra può introdurre ritardi (e quindi imprevedibilità di comportamento) nel momento in cui il processo va a modificare le parti condivise in quanto, almeno nell'istante della modifica iniziale, queste devono essere prima duplicate per poter procedere successivamente alla scrittura dei dati.
- presenza dei kernel thread per ridurre l'overhead connesso alla gestione di attività periodiche che perdurano per l'intero arco di attività del sistema.

4.5 Schedulatore

Lo schedulatore è la componente del kernel che decide quale processo mandare in esecuzione, quando e per quanto tempo; esso ha il compito di distribuire le limitate risorse di tempo-macchina tra i processi presenti nel sistema. Per un ottimale utilizzo di tale risorsa, è necessario che, nell'ipotesi di un numero sufficiente di processi eseguibili, almeno un processo per ciascun core sia effettivamente in esecuzione.

Il problema che lo schedulatore deve risolvere in ogni istante è la scelta di quale processo portare in esecuzione dato un insieme di task possibili. Un sistema operativo multitasking ha la capacità di eseguire in modo inframezzato più di un processo per volta e questo dà all'utente, anche su macchine monoprocessore, l'illusione che più programmi siano eseguiti simultaneamente.

Ulteriori aspetti da considerare sono le condizioni in cui lo schedulatore può effettivamente agire e l'intervallo di tempo necessario ad esplicare ciascun intervento.

Relativamente al primo punto, infatti, se ad esempio si permettesse allo scheduler di operare solo quando viene richiamato esplicitamente dal codice attraverso la chiamata a sistema `schedule()`, potremmo avere programmi in spazio utente che monopolizzano l'intero sistema, continuando ad operare indefinitamente.

Il kernel prevede quindi la presenza di un flag, contenuto nella struttura `thread_info` e quindi dipendente dall'architettura (per x86 è il bit `TIF_NEED_RESCHED` definito in arch/x86/include/asm/thread_info.h), il cui valore viene controllato attraverso la funzione `need_resched()`, che indica se c'è o meno la necessità per lo stesso kernel di attivare quanto prima lo schedulatore; questo parametro viene quindi inizializzato a 1, ad esempio,

quando un processo può essere interrotto o quando viene ad attivarsi un processo avente una priorità maggiore di quello corrente.

L'interrompibilità dei processi gioca un ruolo importante nel definire le situazioni in cui il flag di rischedulazione viene attivato; queste sono suddivise in due gruppi principali:

- **User preemption**: comprendono gli istanti in cui il sistema torna in spazio utente, e dunque generalmente una volta che viene terminata da parte del kernel la gestione di una interruzione o di una chiamata di sistema; corrispondendo ad istanti in cui il sistema è in uno stato consistente, è possibile far intervenire lo scheduler per riprendere il task precedentemente bloccato o portarne in esecuzione uno nuovo, e dunque il flag può essere attivato.

- **Kernel preemption**: nel caso di task operanti in kernel-space la situazione si complica; la prelazione in questo caso è possibile solo se il task che la subisce non detiene nessun lock(sezione 4.8.2). In questo modo, i lock vengono utilizzati per delimitare quelle parti di codice non interrompibili. Poichè uno stesso processo può acquisire più lock, viene previsto all'interno della struttura `thread_info` il campo `preempt_count` che memorizza tale quantità; solo quando questa è nulla il task può subire prelazione.

  In linea generale, la kernel preemption si può avere al termine di una ISR, negli intervalli in cui il kernel torna ad essere interrompibile e quando un task del kernel richiama, esplicitamente o in una situazione di blocco, la funzione `schedule()`.

Per quanto riguarda invece il secondo aspetto, oltre a considerare il tempo impiegato dallo scheduler per indivisuare il prossimo task da mandare in esecuzione, va incluso anche l'itervallo necessario ad effettuare lo scambio tra vecchio e nuovo processo, operazione che viene indicata con il nome di *context switch*.

Essa consta di due parti principali: il passaggio dallo spazio di indirizzamento del vecchio processo a quello del nuovo e il salvataggio dello stato del task precedente, che comprende la memorizzazione dei registri del processore e di altre variabili dipendenti dall'architettura necessarie per poter riprendere l'esecuzione del processo bloccato in un secondo momento. Tale operazione non viene effettuata in un tempo constante: se confrontiamo un context switch tra thread della stessa applicazione e uno tra due task non correlati, vediamo che il primo sarà effettuato in tempi più rapidi in quanto alcune strutture dati sono già condivise dai thread (come lo spazio degli indirizzi) e non è necessario quindi operarne lo scambio.

4.5.1 Parametri di schedulazione

Per definire i parametri di schedulazione da attribuire ai job o ai task il kernel Linux mette a disposizione alcune tipologie di variabili: il timeslice, la priorità e l'affinità cpu.

La *timeslice* è un valore numerico che rappresenta l'intervallo temporale massimo durante il quale un processo può essere eseguito prima di una eventuale preemption. Se troppo lunga si rischia di avere basse prestazioni interattive, se troppo corta si verifica un overhead dovuto all'alta frequenza di cambi di contesto; entrambe le possibilità andrebbero a favorire rispettivamente i processi cpu-bound e I/O bound.

In Linux la dimensione della timeslice dipende sia dal carico di sistema che dal valore di nice associato al task:ad esempio, processi con elevati valori di nice e dunque a bassa priorità ricevono un peso negativo e quindi una porzione minore di timeslice.

Linux riconosce due valori distinti per dichiarare la priorità:

- **valore di nice**:Il valore di nice, tra -20 e + 19 con un valore di default 0, dove a numero maggiore corrisponde una inferiore priorità e quindi una inferiore quantità di tempo cpu dedicata. Il range di valori può essere poi sfruttato secondo strategie diverse a seconda della politica di schedulazione che si vuole implementare. Nei sistemi Unix, e quindi anche in Linux, la priorità è esportata a livello utente grazie al valore di nice.

- **priorità real-time**: I livelli di priorità consentiti di default sono 100, con valori da 0 a 99 e, contrariamente a quanto accade per il valore di nice, a valore maggiore corrisponde priorità più elevata.

  La priorità di tipo real-time si muove in un range compreso tra zero e `MAX_RT_PRIO` (escluso); quest'ultimo ha un valore di default pari a 100, ma rimane comunque un parametro modificabile. I valori tra `MAX_RT_PRIO` e `MAX_RT_PRIO+39` sono invece usati per rimappare i valori di nice, in modo da mantenere completamente separati i due intervalli e permettere ai processi real-time di avere in ogni caso una priorità superiore rispetto a quelli gestiti con politiche non real-time.

Per quanto riguarda il rapporto tra timeslice e valore di nice, possiamo rilevare le seguenti criticità:

- Innanzitutto, occorre predefinire l'ammontare della timeslice associato a ciascun valore possibile del parametro nice, e questo implica un comportamento inferiore all'ottimo, poichè le porzioni sono decise a priori. Ad esempio, se diamo 100ms a priorità 0 e 5ms a priorità +20 otteniamo che, in caso di 2 processi a priorità minima, avremo uno switch ogni 5ms. Una associazione tale sarebbe addirittura controproducente visto che generalmente i processi cpu bound sono a bassa priorità

e, necessitano di essere eseguiti meno frequentemente e per intervalli lunghi.

- Inoltre, vanno considerati i rapporti relativi fra valori di nice; se ad esempio l'intervallo di timeslice decresce linearmente con il valore di nice, ci troveremo con una bassa differenza relativa tra processi di priorità bassa ma con una forte diversità per quelli ad alta priorita.

- Dobbiamo anche fornire timeslice assoluti che il kernel possa misurare come multipli di un qualche temporizzatore interno; questo limita ad esempio il minimo timeslice impostabile, la minima differenza tra timeslice successivi e rende la misura dipendente dalle caratteristiche del temporizzatore stesso.

- Ulteriore problema risiede nella debolezza nei confronti di processi malevoli creati ad hoc per mantenere costantemente la massima priorità e scatenare un problema di starvation per i regolari task del sistema. Alcuni di essi sono risolvibili impostando una crescita di tipo geometrico tra timeslice relativi a valori contigui del parametro nice o disaccoppiando la misurazione del timeslice dai temporizzatori del sistema.Il CFS (sezione 4.5.3) adotta un nuovo approccio che limita gli effetti di questi fenomeni.

Infine, con l'indice di *affinità alla cpu*, il kernel Linux cerca di mantenere l'esecuzione di un processo nel medesimo core in modo da ridurre al massimo eventuali migrazioni che comportano overhead dovuto allo spostamento da una cpu all'altra dei dati relativi al processo stesso.

4.5.2 Parte principale dello schedulatore

Lo schedulatore di Linux, definito in kernel/sched.c, è modulare e permette quindi di utilizzare più algoritmi di schedulazione. Tale modularità è ottenuta attraverso le *classi di schedulazione*, strutture che permettono di raggruppare più tipologie di task ai quali sarà applicato il medesimo algoritmo di scheduling. La porzione di codice principale dello scheduler itera sequenzialmente tra esse (conferendo implicitamente loro una priorità dipendente dalla posizione occupata nella scansione), scegliendo come prossimo processo da mandare in esecuzione quello di priorità maggiore presente nella prima classe non vuota.Ogni classe va a completare l'interfaccia generale offerta dallo schedulatore, implementando i dettagli delle varie funzionalità presenti.

Il kernel attualmente implementa due distinte classi di schedulazione:

- **sched_fair**: nelle varie versioni ha visto cambiare molto spesso l'algoritmo che la governa (attualmente è il CFS) ed è abbinata alle politiche di `SCHED_NORMAL` (lo scheduler time-sharing standard usato per le applicazioni che non richiedono particolari meccanismi real-time),

`SCHED_IDLE` (inserito per processi da eseguire con priorità estremamente basse, minori anche del massimo valore di nice) e `SCHED_BATCH` (pensato per processi cpu-bound) . Questi processi sono abbinati alla priorità data dal valore nice e quindi possono operare prelazione solo su processi appartenenti a questa classe con valore di nice inferiore (ovvero priorità superiore). La priorità dei processi abbinata a questa classe è dinamica e quindi può variare nel corso dell'esecuzione del task j stesso.

- **sched_rt**: mirata ad implementare parzialmente le schedulazioni a priorità fissa incluse nelle drettive POSIX (progettto per la standardizzazione delle API messe a disposizione da varianti di sistemi Unix-like alle applicazioni) e comprendente due politiche differenti:
  - `SCHED_FIFO`: realizza un semplice algoritmo di schedulazione fifo. I processi di questa classe hanno sempre la priorità su quelli normali e possono subire prelazione solo in tre casi: quando rilasciano volontariamente la cpu, si bloccano o si rende disponibile un processo della stessa classe a priorità maggiore. I processi con uguale priorità sono gestiti in modo sequenziale passando dall'uno all'altro solo nei primi due dei tre casi appena citati. Diversamente dalla precedente, opera con priorità statiche, offrendo quindi minore flessibilità ma una maggiore aderenza a rispettare l'ordine stabilito a priori (un processo a priorità maggiore rimane tale e quindi anche la prevedibilità dei parametri temporali di esecuzione risulta maggiore).
  - `SCHED_RR`: simile al precedente con l'eccezione che l'esecuzione del processo è regolata anche da un timeslice, al termine del quale va ridefinito quale processo mandare in esecuzione. In questo caso, quindi, il singolo task può essere interrotto sia da processi a priorità maggiore sia nel momento in cui il timeslice si esaurisce.

Il real-time fornito dai due schedulatori real-time è di tipo soft, in quanto il kernel non garantisce il rispetto assoluto dei vincoli temporali, poichè la sue strutture standard non permettono di registrare per ogni processo la rispettiva deadline (la stessa struttura del descrittore di processo ha alcuni campi per gestire la priorità ma nessuno per registrare il valore della deadline associata al job del task).

Tali politiche non sono però particolarmente sofisticate e questo costituisce un problema se si progettano applicazioni sensibili ai vincoli temporali, in modo particolare in ambito embedded in quanto, per sopperire ai limiti particolarmente stringenti sulle quantità di risorse disponibili, è indispensabile sfruttarle nella maniera più efficiente possibile.

Alcuni ricercatori [12] hanno promosso l'implementazione, sfruttando il framework altamente modulare offerto dal kernel, di ulteriori classi di schedu-

lazione quali la *sched_ edf*, con la quale si è tentato di inserire una politica di scheduling che vede la combinazione dell'algoritmo EDF con un meccanismo di *resource reservation* basato su CBS.

Malgrado sia stato presentato ancora nel 2009 e sia tutt'ora in continuo sviluppo [22](terminata da poco la settima versione) il processo per l'inserimento nel kernel ufficiale è ancora in divenire (rimane comunque possibile integrarlo tramite patch).

4.5.3 Algoritmi di schedulazione in Linux

Dalla prima versione del 1991 fino alla 2.4 gli schedulatori disponibili per Linux erano molto semplici ma scalavano in modo pessimo su macchine con molti processi attivi o dotate di numerose cpu. Durante lo sviluppo della versione 2.5 viene introdotto lo scheduler O(1) che, introducendo un algoritmo a tempo costante per il calcolo delle timeslice e delle code di esecuzione per ciascun processore presente, va ad eliminare molti dei punti deboli degli scheduler precedenti. Con il tempo anche questo ha evidenziato alcune limitazioni, legate specialmente alla gestione di processi sensibili ai tempi di latenza, come ad esempio quelli ad alta interazione con l'utente. L' O(1), dalle buone prestazioni su macchine server (in cui i processi interattivi costituiscono un'esigua minoranza), mostra quindi il fianco quando viene impiegato in sistemi fortemente dedicati all'interazione, quali desktop o sistemi embedded legati all'elettronica di largo consumo. Con la versione 2.6 si sono viste varie proposte di nuovi scheduler, tra i quali il Rotating Staircase Deadline, che introduce il concetto di schedulazione equa e che servirà poi da base all'implementazione del definitivo Completely Fair Scheduler (CFS) introdotto dalla versione 2.6.23.

CFS

Il Complete Fair Scheduler è basato sul concetto di cpu multitasking ideale, capace cioè, in un qualsiasi intervallo di tempo, di servire n processi garantendo a ciascuno una porzione di tempo processore pari ad $1/n$ dell'intervallo dato.

La definizione teorica possiede due vincoli impossibili da riprodurre perfettamente nella realtà: in ogni istante ciascun core potrà eseguire comunque un solo processo ed inoltre non è possibile coprire tutti i possibili intervalli poichè la cpu ha una risoluzione temporale minima data dalla frequenza del suo clock; inoltre il passaggio da un processo al successivo non è istantaneo ma occupa esso stesso un certo intervallo di tempo. La realizzazione dell'algoritmo soffrirà dunque di un certo grado di approssimazione.

L'obiettivo dell'algoritmo è quello di servire tutti i processi presenti in modo equo, garantendo a ciascuno una porzione di tempo cpu pesata in base al valore di nice.

Questo viene realizzato con il seguente criterio:

- viene definito come parametro dell'algoritmo di scheduler una timeslice fissa (*targeted latency*) scelta in base al grado di approssimazione del comportamento ideale che si vuole ottenere: ad esempio, nel caso vi siano 10 processi attivi di pari priorità e la timeslice sia pari a 20ms, ogni processo verrà eseguito per al massimo 2ms.

  In presenza di numerosi processi tale timeslice potrebbe ridursi eccessivamente innalzando in modo inaccettabile l'overhead dovuto agli switch; per evitare ciò, se ne fissa un valore minimo detto *granularità minima*, generalmente impostata ad 1 ms (l'equità di CFS quindi tende a decadere progressivamente non appena il numero di processi determina l'utilizzo del valore minimo invece di quello effettivamente calcolato).

- per ogni processo attivo viene abbinato un peso [27] w_i calcolato in funzione del valore di nice tramite l'equazione $w_i = 1024 * 1.25^{(-nice)}$; come si vede, a priorità maggiore (valore di nice minore) viene associato un peso maggiore. Inoltre, si realizza così una crescita del peso in scala geometrica.

- sempre per ciascun processo nella coda ready viene è presente la variabile `vruntime`, in cui viene memorizzato il tempo effettivo di esecuzione, normalizzato poi in base al rapporto tra peso associato ad un task con valore di nice 0 e il peso del task corrente; in questo modo, i processi a priorità maggiore ($nice < 0$) hanno un `vruntime` minore, e viceversa.

- il valore di `vruntime` viene aggiornato periodicamente oppure negli istanti in cui un task effettua una transizione di stato da eseguibile a bloccato (o viceversa). Ad ogni aggiornamento, l'algoritmo manda in esecuzione il task associato al minimo valore della `vruntime`; è proprio il valore di quest'ultima, quindi, a definire la priorità del task in base alla politica adottata dal CFS.

In questo modo, le uniche `vruntime` che aumentano durante l'aggiornamento sono quelle dei processi correntemente eseguiti; questo permette quindi ai restanti task in stato ready (la cui `vruntime` rimane costante) di avere, con il passare del tempo, maggiori probabilità per essere scelti come nuovi processi da eseguire e quindi prelazionare quelli correnti.

Si può notare che, se un processo trascorre molto tempo in attesa e quindi è presente in modo sporadico nella coda ready (come accade per molti processi interattivi), il suo `vruntime` crescerà lentamente , mantenendo quindi per un certo periodo un valore basso e dunque conferendo al processo un più elevato livello di priorità.

Per tenere traccia dei processi e delle porzioni di tempo cpu impiegato viene usato un albero rosso-nero che, grazie alla sua proprietà di autobilanciamento e alla politica di gestione dei suoi elementi, permette un accesso all'elemento con `vruntime` minore (che occupa il nodo all'estrema sinistra dell'abero) in un tempo $O(\log(n))$ e quindi debolmente influenzato dal numero di processi presenti.

Nel dettaglio, l'albero rosso-nero è un particolare albero di ricerca binario auto-bilanciante, che deve soddisfare i seguenti vincoli:

1. Tutti i nodi possono essere rossi o neri.
2. I nodi foglia sono neri e non contengono dati.
3. Tutti i nodi, escclusi i nodi foglia, hanno due figli.
4. Se un nodo è rosso, entrambi i suoi figli devono essere neri 5. Il percorso da un nodo ad una delle sue foglie contiene lo stesso numero di nodi neri del percorso più breve ad una qualsiasi delle sue foglie

La sua implementazione in Linux (ritrovabile nei file lib/rbtree.c e nell'header linux/rbtree.h) non fornisce metodi di ricerca e inserimento, poichè il linguaggio C non facilita la programmazione generica e gli sviluppatori sono concordi nel ritenere più efficiente una implementazione di tali procedure legata al particolare contesto di funzionamento e quindi realizzata ad hoc all'interno di ciascun progetto.

BFS

Tra le patch proposte in ambito soft real-time spicca la proposta di un algoritmo di scheduling alternativo a CFS,il *Brain fuck scheduler (BFS)* [24], che nelle intenzioni dell'autore è stato progettato specificatamente per macchine con potenzialità elaborative limitate e per fornire una latenza particolarmente bassa ai programmi fortemente interattivi, senza l'utilizzo di euristiche particolari i cui effetti sono generalmente difficili da prevedere.

Adottato solo da alcune distribuzioni [25] [26], l'algoritmo utilizza una coda unica per tutti i processi del sistema, rimuovendo la necessità di utilizzare complessi meccanismi di bilanciamento di carico (necessari in caso si utilizzino strutture per-cpu) e quindi mirando a contenere l'intervallo di tempo necessario alle computazioni insite nello scheduler stesso.

L'algoritmo [29] associa a ciascun task una timeslice e una deadline virtuale (virtuale perchè non se ne garantisce il rispetto) la cui entità è pesata sul valore di nice (si parte da un valore di base per nice=-20 e lo si incrementa di un 10% per ogni livello di nice successivo). Nella lista vengono mantenuti solo i processi ready e non quelli correntemente attivi, in quanto i tempi di esecuzione di questi ultimi sono mantenuti localmente dalle cpu. Quando un nuovo processo viene inserito, si controlla se esistono cpu in idle

su cui può essere eseguito o se la sua deadline gli permette di prelazionare uno dei processi attualmente in corso (il controllo occuperà un tempo $O(N)$ con N numero delle cpu di sistema).

La propensione a favorire i processi interattivi è insita nelle modalità di aggiornamento della deadline virtuale: se il processo termina la propria timeslice, viene riportato nella coda globale con una nuova deadline virtuale; se invece questo si pone in stato di sleep per un motivo qualsiasi viene posizionato in coda mantenendone il valore attuale (la sua deadline virtuale avrà quindi maggiore probabilità di essere una delle più prossime temporalmente).

Alcuni test condotti [14] [28] evidenziano la minore latenza e le superiori prestazioni interattive dello scheduler BFS, a fronte di un comportamento peggiore per quanto riguarda i tempi di esecuzione complessivi dei programmi rispetto al CFS: ne viene confermata quindi la maggiore adattabilità ad operare in ambito soft real-time.

4.6 Misura del tempo

Il concetto di tempo per un computer è di fondamentale importanza in quanto molte attività che questo svolge devono attuarsi con una data periodicità: l'aggiornamento dell'ora di sistema e dei parametri statistici di utilizzo del processore e delle risorse da parte dei processi, il bilanciamento delle code di esecuzione nelle architetture multiprocessoree la misurazione di intervalli di ritardo.

La misurazione del tempo si fonda su un meccanismo hardware detto *system timer*, il quale genera un'interruzione con una periodicità preprogrammabile dal kernel, il *tick rate*, definito nel file /include/asm-generic/param.h che riportiamo qui sotto:

```
#ifndef __ASM_GENERIC_PARAM_H
#define __ASM_GENERIC_PARAM_H

#include <uapi/asm-generic/param.h>

# undef HZ
# define HZ CONFIG_HZ /* Internal kernel timer frequency */
# define USER_HZ 100 /* some user interfaces are */
# define CLOCKS_PER_SEC (USER_HZ) /* in "ticks" like times() */
#endif /* __ASM_GENERIC_PARAM_H */1
```

La variabile `HZ` contiene il valore del tick rate assegnato; come si può notare andando a vedere l'header incluso alla prima riga, essa è inizializzata di default al valore di 100Hz ma può essere modificata tramite `CONFIG_HZ`, uno dei parametri definibili al momento della compilazione tramite il menù di configurazione del kernel, che consente l'inserimento di valori pari a 250,

300 e 1000Hz. Il numero di tick trascorsi dall'accensione del sistema viene poi memorizzata nella variabile jiffies.

La scelta del valore più opportuno va effettuata considerando i seguenti aspetti:

- Un valore elevato permette una maggiore risoluzione e accuratezza nel determinare le misure temporali; ad esempio, per quanto riguarda la schedulazione, si possono definire con maggior precisione gli istanti in cui deve essere effettuata la preemption di un task. Inoltre, si possono ottenere tempi di latenza dello scheduler più ridotti: poniamo il caso che ad un task restino ancora 2ms di timeslice disponibili, trascorsi i quali lo schedulatore dovrebbe attivare la preemption; se si ha un tick rate pari a 100Hz, questo sarà possibile solo dopo 10ms, comportando un ritardo nella rischedulazione di 8ms, mentre se il tick rate fosse stato impostato a 1000Hz il ritardo si sarebeb ridotto notevolmente (mediamente pari a 0.5ms).

- Un elevato tich rate comporta però anche diversi svantaggi: poichè gli interrupt sollevati sono gestiti tramite una funzione fornita dal kernel, il *timer handler* (funzione che si occupa di attivare tutte le attività periodiche definite), questa dovrà intervenire con maggiore frequenza, causando quindi un maggiore overhead; inoltre, questo causa un più frequente aggiornamento della cache della cpu, con un conseguente incremento dei consumi energetici.

Per ovviare in parte alle problematiche relative ai consumi, il menù di configurazione per la compilazione del kernel mette a disposizione, nel sottomenù `General setup --> Timers subsystem` l'opzione `Tickless system`; attivandola, il kernel schedula l'intervento del gestore del timer non più in modo periodico ma basandosi sulle scadenze degli altri timers presenti nel sistema; questo comporta sia una riduzione dell'overhead che della potenza dissipata, in quanto il gestore interviene solo quando necessario (la cosa è più evidente se il sistema è in stato idle per lunghi intervalli).

Se si desiderano risoluzioni superiori [6] sono a disposizione ulteriori strumenti:

- **Time stamp counter (TSC)**: è un registro messo a disposizione da cpu quali le x86 il cui valore viene incrementato di una unità ad ogni clock del processore, ed è accessibile tramite la funzione `rdtsc`. La risoluzione che può offrire dipende ovviamente dalla frequenza della cpu (essendo generalmente nell'ordine del gigahertz, si ottengono incrementi del contatore in intervalli di 1ns).

- **APIC locale alla cpu**: l'*Advanced programmable interrupt controller* è simile al system timer, con la differenza che emette segnali di interruzione indirizzati esclusivamente alla cpu locale, è basato sulla frequenza

di bus ed è programmabile per essere aggiornato ogni 1,2,4,8,16,32, 64 o 128 segnali di clock della stessa (sempre in riferimento ad architetture x86)

- **High precision event timer (HPET)**: supportato dalla versione 2.6 di Linux, fornisce fino a 8 contatori a 32 o 64 bit indipendenti, incrementati ad una frequenza minima di 10Mhz (e quindi al massimo ogni 100ns); ciascun contatore è poi associabile fino a 32 timers diversi, costituiti da un *match register* in cui viene impostato l'intervallo da attendere e un comparatore che confronta il valore del contatore con quello del registro per determinare lo scadere dell'attesa.

Oltre alla misurazione del tempo, il kernel offre svariati strumenti per definire intervalli di attesa. I principali che si possono impiegare sono:

- **Kernel timers**: sono rappresentati dal kernel attraverso la struttura `struct timer_list`, definita in linux/timer.h, di cui riportiamo le righe principali:

```
  struct timer_list {
/*
 * All fields that change during normal runtime
 * grouped to the same cacheline
 */
struct list_head entry;
unsigned long expires;
struct tvec_base *base;

void (*function)(unsigned long);
unsigned long data;
...

};
```

.

Per ciascuno timer può essere impostato il periodo di ritardo (espresso in jiffies, ovvero in numero di tick) e la funzione che dovrà essere eseguita allo scadere di questo. Più precisamente, tale funzione viene eseguita non appena il valore della variabile di sistema jiffies è uguale o maggiore al ritardo impostato, e quindi si avrà sempre un errore legato al valore di `HZ` impostato dal sistema, la cui risoluzione è relativamente bassa e risulta quindi un parametro critico se si devono supportare applicazioni soft o hard real-time; se ad esempio avessimo `HZ=100` si potrà avere un errore medio di 5ms.

Il kernel esegue i timer in contesto di softirq (vedi sezione 4.7.3). Tutti i timer sono conservati in una lista ma, per migliorarne la gestione, essi sono suddivisi in 5 gruppi definiti in base al tempo mancante alla loro scadenza e quindi, ad ogni aggiornamento, scalano di gruppo in gruppo in modo che il kernel debba impiegare un tempo minore per individuare quelli che sono effettivamente in scadenza.

- **Busy looping**: utilizzato quando il ritardo deve essere pari ad un multiplo del tick rate e la precisione temporale è un parametro secondario; poichè implementano un loop di tipo busy comportano un forte dispendio del tempo cpu e sono quindi usati raramente.

- **Ritardi brevi**: utili per impostare ritardi più piccoli del periodo di tick. Il kernel mette a disposizione tre diverse funzioni, `mdelay`, `udelay` e `ndelay`, definite in linux/delay.h, alle quali si passa un parametro di tipo long pari rispettivamente al numero di millisecondi, microsecondi o nanosecondi del ritardo desiderato. Poichè i tempi desiderati sono inferiori generalmente al tick, l'intervallo viene misurato ricorrendo al *BogoMIPS*, un valore che indica il numero di iterazioni vuote che il processore è in grado di realizzare in un dato periodo (generalmente un tick), e la precisione della misura dipende quindi dalle caratteristiche dell'hardware presente. Come per le altre soluzioni in busy waiting, se ne consiglia l'utilizzo solo per intervalli brevi per non sprecare eccessiva potenza elaborativa.

- **Funzione** `schedule_timeout()`: permette di porre un task in stato di sleep fino a che non è trascorso l'intervallo specificato. In realtà anche in questo caso la precisione è molto bassa perchè basata sul tick rate e perchè viene garantito il risveglio del task dopo un lasso di tempo uguale o maggiore a quello dichiarato, ed inoltre non può essere utilizzata in contesto di interruzione.

4.7 Interruzioni

L'hardware di un calcolatore possiede velocità di esecuzione di vari ordini di grandezza inferiori a quelle delle cpu, e quindi al kernel non conviene lanciare richieste verso le periferiche e porsi in attesa della risposta, poichè ciò comporterebbe un enorme spreco di tempo di calcolo. I modi alternativi per gestire tale comunicazione sono sostanzialmente due:

- **Polling**: il kernel controlla periodicamente lo stato della periferica per verificarne ventuali modifiche e agire di conseguenza. Questo comporta overhead poichè i controlli vengono ripetuti periodicamente indipendentemente dal fatto che la periferica sia pronta o meno.

- **Interruzioni**: permettono all'hardware di segnalare al processore, attraverso l'invio di un segnale elettrico, l'istante in cui i risultati sono disponibili. A sua volta, il processore interrompe l'esecuzione corrente per segnalare al kernel la ricezione del segnale e permettere al sistema operativo di rispondere opportunamente all'evento. Attraverso un valore univoco (IRQ) associato a ciascun interrupt, questi ultimi possono essere associati alle differenti periferiche presenti e questo permette al sistema operativo di conoscere quale periferica ha sollevato una determinata interruzione; nei classici pc desktop, ad esempio, l'IRQ 0 è associato al timer di sistema e l'1 alla tastiera, mentre altri valori possono essere associati dinamicamente alle linee di interruzione durante l'avvio del sistema.

4.7.1 Gestori di interruzioni

Le funzioni del kernel adibite a gestire le interruzioni sono gli *interrupt handler*, chiamate anche *interrupt service routine (ISR)*; ogni dispositivo che può generare interruzioni ne ha una associata, il cui codice è contenuto all'interno di un driver (componente del kernel che definisce le funzioni con cui questo può comunicare con la particolare periferica); in LInux, poi, tale funzione deve rispettare un prototipo predefinito, in modo tale che il kernel possa gestirle attraverso un'interfaccia comune.

Requisiti fondamentali del gestore sono che risponda rapidamente all'evento di interruzione e che la sua esecuzione richieda tempi minimi, in quanto essendo asincrono va ad interrompere in modo totalmente imprevedibile le altre attività del sistema; le operazioni che dovrebbe limitarsi a svolgere sono quelle temporalmente critiche, come avvisare il dispositivo dell'avvenuta ricezione dell'interrupt o resettarne lo stato.

La seconda proprietà a volte è difficile da realizzare in quanto può essere richiesta anche una manipolazione dei dati ricevuti; si pensi ad esempio alla ricezione di un pacchetto dati dalla rete, che va copiato in memoria e i dati devono subire varie rielaborazioni prima di essere fruiti dal sistema.

Una soluzione a questo si è trovata dividendo l'azione del gestore in due parti: la top half, nella quale vengono svolte le operazioni temporalmente critiche (come segnalare l'avvenuta ricezione del segnale), mentre la parte restante viene affidata alla bottom half, che può essere eseguita in un secondo momento, quando il carico di sistema lo consente. Vediamo ora quali strumenti Linux mette a disposizione per implementare queste due porzioni dei gestori di interruzioni.

4.7.2 Top half

Se un dispositivo è in grado di generare interruzioni, il suo driver deve registrare un gestore nel sistema, attraverso la funzione `request_irq()`, definita

in linux/interrupt.h e il cui prototipo riportiamo per comodità:

```
request_irq(unsigned int irq, irq_handler_t handler,
    unsigned long flags, const char *name, void *dev)
```

I parametri di maggiore sono i primi tre, con cui si indicano rispettivamente il numero che individua univoccamente l'interruzione, la funzione che agirà da gestore e una maschera di bit per abilitare o disabilitare le varie interruzioni; quest'ultimo può assumere solamente alcuni valori predefinti in linux/interrupt.h, di cui i più significativi sono:

- `IRQF_DISABLED`:permette di disabilitare tutte le interruzioni mentre veniva eseguito un interrupt handler (mentre di default si disabilita solo l'interruzione gestita). Nel file stesso in cui è definito si specifica però l'intenzione degli sviluppatori del kernel di eliminarlo.

- `IRQF_SHARED`: si specifica che l'interruzione è condivisa da più periferiche (quindi dovrà essere impiegato l'ultimo dei parametri di `request_irq()` per definire quale dispositivo ha generato l'interrupt):

La sequenza di eventi sottesi a questa prima fase della gestione di un'interruzione possono essere così riassunti:

- Una periferica genera il segnale di interrupt.

- Appena giunge alla cpu, questa blocca qualsiasi task in esecuzione su di essa, disabilita le interruzioni e carica il codice presente in una locazione predefinita della memoria(*entry point*).

- viene salvato lo stato del processo interrotto e il numero identificativo della linea di interruzione e viene richiamata la funzione `do_IRQ()` che si occupa sia di segnalare al dispositivo l'avvenuta ricezione del segnale che di disabilitare l'ulteriore ricezione di interrupt sulla linea interessata.

- viene verificata poi la presenza di un gestore valido; se questo esiste, viene prima ripristinato il sistema di interruzioni precedentemente disabilitato dalal cpu (a meno che non sia specificato nel gestore il parametro `IRQF_DISABLED`) e poi eseguito il gestore, al termine del quale sono nuovamente disabilitate le interruzioni.

- infine, il sistema controlla se c'è bisogno di una rischedulazione attraverso la funzione `need_resched()`; in caso affermativo, lo schedulatore viene immediatamente attivato a meno che il processo interrotto non fosse in kernel-space e avesse un `preempt_count` diverso da zero (e quindi non è ammesso interrompere il task attraverso una rischedulazione).

Il funzionamento stesso della top half dimostra che le interruzioni, in Linux, sono eseguite con precedenza massima, in quanto determinano il blocco di qualsiasi processo in esecuzione, indipendentemente dal suo grado di priorità. Questo, se da una parte garantisce un tempo di risposta minore nelle comunicazioni con le periferiche, introduce una ulteriore causa di variabiità nella deterrminazione dei tempi di esecuzione effettivi dei task del sistema.

4.7.3 Bottom half

La bottom half è introdotta perchè il gestore di interruzione nella top half lavora con almeno la propria linea di interruzione disabilitata su tutti i processori, ma è anche possibile imporre la disabilitazione di tutti gli IRQ locali alla cpu; quiindi per motivi prestazionali (non si possono ricevere altri segnali di interruzione sulla stessa linea fino a quando la top-half non ha terminato la sua esecuzione) deve essere mantenuto con un tempo di esecuzione minimale.

Il compito della bottom half è differire gran parte del lavoro in un secondo momento quando le interruzioni saranno ripristinate e il sistema meno occupato in altre attività. Diversamente dalla top-half, vi sono più meccanismi forniti nel tempo dal kernel Linux per realizzarla. Il primo meccanismo prevedeva una interfaccia BH che forniva una lista di 32 bottom half statiche, ciascuna delle quali sincronizzata globalmente; di conseguenza non poteva essere eseguita più di una bottom half per volta nell'intero sistema, anche se multiprocessore, e questo costituiva un vincolo deleterio per le prestazioni. Successivamente si sono introdotte le task queue, che contenevano liste di funzioni da chiamare; i driver registravano la propria bottom half in una di queste liste che, ad un certo momento, venivano sequenzialmente attraversate per eseguire le funzioni ivi memorizzate. Con il kernel 2.3 vengono introdotte softirq e tasklet, due mecccanismi che finalmente vanno a sostituire completamente le vecchie BH, mentre dal 2.5 le workqueue vanno a sostituire le task queue.

Softirq

Le softirq sono un insieme statico di bottom half che possono essere eseguite simultaneamente su qualsiasi processore, anche nel caso siano dello stesso tipo. Sono usate nelle parti di codice in cui le prestazioni sono critiche ma il loro uso richiede accortezza proprio per la loro caratteristica di esecuzione concorrente. Rappresentate dalla struttura `softirq_action` definita in linux/interrupt.h, mentre in kernel/softirq.c è dichiarato un array statico che permette un massimo di 32 di tali strutture(nel 3.7.6 ne sono definite dieci); l'indice della posizione all'interno dell'array definisce anche le relative prorità (quindi softirq in testa all'array possiedono una priorità maggiore di quelle che seguono).

Eccettuati i gestori di interruzioni, nessuna altra struttura può causare prelazione sulle softirq, nemmeno un'altra softirq. Prima di poter essere eseguita una softirq deve essere marcata, e questo viene generalmente fatto come ultimia operazione del gestore di interruzioni. Tutte le softirq pendenti (marcate ma non ancora eseguite) sono controllate ed eseguite o al ritorno da una interruzione hardware, da parte del thread `ksoftirqd` o in qualsiasi punto in cui il codice le richiama esplicitamente. Data la loro natura, se utilizzano risorse esse vanno adeguatamente protette con un lock (ricordiamo che lo stesso tipo di softirq può essere eseguita simultaneamente da più processori).

Tasklet

Le tasklet sono costruite sulla struttura delle softirq; tasklet di tipo differente possono essere eseguite in modo concorrente, mentre alle tasklet dello stesso tipo ciò non è consentito. Rispetto alle softirq rappresentano un compromesso tra prestazioni e maggiore facilità di utilizzo nella programmazione, aggiunta alla possibilità di registrarle dinamicamente. Esse sono sostanzialmente rappresentate da due softirq: `HI_SOFTIRQ` e `TASKLET_SOFTIRQ`, dove la prima ha priorità maggiore della seconda.

La struttura con cui le tasklet sono rappresentate (in linux/interrupt.h) contiene il prototipo della funzione che agirà da gestore della tasklet, a cui è possibile passare un solo parametro di tipo long, una variabile di stato che segnala se la tasklet è attualmente schedulata ma non ancora in esecuzione o se è in esecuzione (assume tre stati possibili: zero, `TASKLET_STATE_SCHED` e `TASKLET_STATE_RUN`) e una variabile count che, se non nulla, indica che la tasklet è disabilitata e non può essere eseguita. I due tipi di tasklet visti sono schedulati in due differenti strutture, `tasklet_vec` e `tasklet_hi_vec`, che il kernel mette a disposizione a coppie per ciascun processore.

La funzione che opera da gestore della tasklet, essendo quest'ultima costruita sulle softirq, non deve utilizzare sottofunzioni che possono passare in stato di sleep e, lavorando con le interruzioni abilitate, vanno sempre protetti con meccanismi di lock i dati eventualmente condivisi con altri gestori di interruzioni o con tasklet di tipo diverso. Il problema non si pone invece per tasklet dello stesso tipo, poichè la loro stessa politica di funzionamento ne impedisce una esecuzione simultanea e quindi una possibile interferenza nella manipolazione dei dati condivisi. Per schedulare effettivamente la tasklet si usa la funzione `tasklet_schedule()`, alla quale va passato come unico parametro un puntatore alla struttura che definisce la tasklet stessa.

Sia le softirq che le tasklet sono aggiunti ad un insieme di thread del kernel che ne gestiscono l'esecuzione nei momenti in cui il sistema è sovraccarico di richieste di esecuzione di softirq. Questo accade se ad esempio le periferiche inviano con frequenza elevata segnali di interruzione o anche nel caso che una softirq rimarchi se stessa per una successiva esecuzione; in tali situazioni,

è possibile che le applicazioni in spazio utente vadano a soffrire di starvation, ma d'altro canto risulta inaccettabile non elaborare le softirq che si autoriattivano in maniera temporizzata.

Le soluzioni proposte sono state varie:

- eseguire le softirq nell'ordine in cui arrivano e ricontrollare subito dopo la presenza di altre softirq sollevate ed eseguirle subito. Questo permette un controllo temporizzato ma in caso di elevato regime di richieste le applicazioni utente possono essere rallentate in modo inaccettabile

- Eseguire le softirq senza comprendere però quelle che vengono riattivate, le quali quindi verranno effettivamente elaborate solo al controllo successivo e quindi generalmente al successivo presentarsi di un'interruzione, la qual cosa può significare un intervallo di attesa non prevedibile nella sua massima lunghezza. In questo modo si ha il risultato opposto al precedente, con il problema di una non immediata elaborazione delle softirq.

La soluzione attualmente implementata costituisce una sorta di compromesso tra le precedenti: in caso di una crescita eccessiva del numero di softirq, il kernel va a risvegliare una famiglia di kernel thread, uno per ciascuna cpu, che si prendono carico delle softirq in eccesso e, essendo eseguiti alla priorità più bassa (nice a +19), le andranno ad eseguire senza ostacolare attività di maggiore importanza. Questo permette in un sistema generalmente in stato idle di rispondere subito alle softirq e ad un sistema sovraccarico di rispondere comunque alle softirq in eccesso, ovviamente con tempi generalmente più lunghi di quelli attesi. I thread in questione si riconoscono dal nome costituito dalla stringa `ksoftirqd/` seguita da un numero progressivo (coincidente con l'identificativo della cpu alla quale sono associati).

Workqueue

Le *workqueue* sono un meccanismo che differisce l'elaborazione affidandola direttamente a thread del kernel la cui bottom half viene eseguita sempre in un contesto di processo, e quindi è schedulabile e può utilizzare anche funzioni che prevedono la possibilità di porsi in stato sleep.

Risultano quindi ottimali se si devono allocare grandi quantità di memoria, attendere l'acquisizione di un semaforo o fornire funzionalità di I/O bloccante. I kernel thread ad esse associati sono detti *worker thread* e sono riconoscibili dalla stringa events/ seguita come per le `ksoftirqd` dal numero del processore a cui sono abbinati.

Al di là dell'intero funzionamento, resta da evidenziare che, pur essendo eseguito in un contesto di processo, il gestore di workqueue non può accedere alla memoria in spazio utente poichè i kernel thread non hanno associata una mappa della memoria user-space. La protezione dei dati e dall'interferenze

tra workqueue e resto del kernel è gestita come quella di un normale processo in spazio utente e quindi non richiede l'utilizzo di meccanismi di locking, semplificando l'utilizzo di tale strumento. Il lavoro delle workqueue viene eseguito ad ogni risveglio del rispettivo thread che le gestisce; per assicurarsi invece che le funzioni di bottom half ancora pendenti vengano completate prima di proseguire nell'esecuzione è disponibile una specifica funzione, `flush_scheduled_work`. E' inoltre possibile schedulare alcuni lavori imponendo che l'inizio della loro esecuzione attenda almeno un ritardo minimo specificato.

Raffronto implementazioni della bottom half

In conclusione, i tre meccanismi visti presentano le seguenti proprietà:

- le softirq devono prevedere meccanismi di protezione dei dati per assicurarne una corretta condivisione ma grazie alla capacità di essere eseguite concorrentemente su più processori sono l'alternativa migliore, specialmente nel caso di codice fortemente basato sull'utilizzo di thread, per utilizzi intensi e temporalmente critici.

- Le tasklet si rivelano migliori invece se usate in codice con un basso o nullo utilizzo di thread, poichè l'impossibilità di eseguirne contemporaneamente più dello stesso tipo semplifica la gestione della sincronizzazione.

- Le work queue rimangono invece la sola alternativa nel caso il codice necessiti di essere eseguito in contesto di processo, ad esempio nel caso si debbano forzatamente utilizzare funzioni bloccanti.

4.8 Sincronizzazione

Come già illustrato nella prima parte, un aspetto che va ad incidere sulle prestazioni in tempo reale riguarda l'accesso sincronizzato a risorse o strutture dati del sistema.

In modo particolare essa è legata alla *scalabilità*, ovvero alla capacità del sistema di aumentare le proprie prestazioni in modo proporzionale alla quantità di risorse disponibili. Questo vale in misura maggiore per sistemi server molto estesi, costituiti generalmente da centinaia o migliaia di cpu, mentre per i sistemi embedded tale fattore incide sicuramente in misura minore ma deve essere considerato [8], in quanto il contenimento dei consumi in questi dispositivi viene ricercato preferendo l'aumento del numero di core all'innalzamento delle frequenze operative.

La presenza di strutture dati sincronizzate, infatti, può divenire un pesante collo di bottiglia per le prestazioni nel momento in cui il numero di

processi che vi accede supera una determinata soglia, innescando tempi di attesa inaccettabili.

Oltre alle problematiche già viste, che incidevano sull'imprevedibilità dei tempi di esecuzione, va aggiunto il fenomeno delle *race conditions*, ovvero situazioni nelle quali lo stato di una variabile condivisa diventa dipendente dall'ordine delle operazioni di accesso eseguite dai vari thread che la utilizzano; queste, pur non incidendo sulle tempistiche, danno luogo ad errori nei risultati forniti.

All'interno del kernel, sono vari i meccanismi che innescano problematiche di concorrenza:

- **Interrupt**: abbiamo visto che possono interrompere in modo asincrono un qualsiasi processo in esecuzione.
- **Softirq e tasklet**: istanze multiple possono essere eseguite simultaneamente ed accedere a risorse condivise.
- **Prelazione del kernel**: i processi del kernel possono interrompersi vicendevolmente.
- **Sincronizzazione con lo spazio utente**: task del kernel possono porsi in stato di sleep e richiamare un nuovo processo.
- **Presenza di più cpu o core**: è possibile che una stessa porzione di codice kernel sia eseguita simultaneamente da più core.

Il kernel Linux offre, per la gestione delle più disparate forme di sincronizzazione, una vasta gamma di strumenti , di cui andremo a presentare le principali caratteristiche.

4.8.1 Operazioni atomiche

Le operazioni atomiche corrispondono ad un insieme di funzionalità eseguite sempre senza poter essere interrotte. Queste diventano fondamentali, ad esempio, per risolvere i casi di race condition. Poniamo il caso che molti thread condividano una sola variabile i, inizialmente nulla, il cui compito è memorizzare il numero di accessi effettuati dai thread, e ipotizziamo che le operazioni di lettura e di modifica del valore corrente siano eseguite separatamente; in queste ipotesi si possono verificare sequenze che comportano un errato aggiornamento della variabile, come quella ora indicata:

- il thread τ_1 accede ad i e ne legge il valore (0).
- il thread τ_2 accede ad i e ne legge il valore (0).
- τ_1 incrementa i al valore 1.
- τ_2 incrementa i al valore 1.

Ne risulta dunque un errato conteggio degli accessi, risolvibile solo inglobando le operazioni di lettura e modifica di ciascun thread in una sola funzione non interrompibile.

L'influsso di tale meccanismo si limita però alla sola correttezza dei risultati, senza incidere sui tempi di esecuzione; inoltre, è utilizzabile solo per variabili semplici.

4.8.2 Spin locks

Nel caso si debbano sincronizzare strutture più complesse, una possibile soluzione è offerta dagli *spin locks*, ossia strutture che possono essere allocate al più ad un singolo thread di esecuzione per volta; se un secondo thread tenta di accedervi mentre sono già state assegnate, questo si pone in attesa di tipo busy waiting fino a che il lock non ritorna disponibile. Ne consegue che questo tipo di struttura va utilizzata per dati che vengono allocati a ciascun task per tempi molto brevi, in quanto la tipologia di attesa implementata comporta uun forte spreco di capacità elaborativa della cpu.

Inoltre, essendo funzioni non bloccanti, possono essere utilizzate all'interno dei gestori di interruzioni.

All'interno degli spin locks si distinguono quelli di tipo *reader-writer (rw)*, impiegati nella gestione degli accessi nel caso i thread interessati possano essere divisi in due categorie ben distinte: i lettori, che accedono alla variabile condivisa esclusivamente per leggerne il valore, e gli scrittori, che invece concorrono alla sua modifica. A ciascuna tipologia vengono abbinate specifiche operazioni di allocazione e rilascio del lock: gli accessi in lettura, non modificando il valore dei dati, possono essere realizzati simultaneamente, mentre quelli in scrittura sono effettuati in mutua esclusione.

Nell'implementazione presente in Linux, inoltre, deve essere evidenziato il fatto che i processi reader sono sempre privilegiati rispetto ai writer; infatti, se un processo scrittore è in attesa della disponibiilità del lock, detenuto da uno o più task lettori, eventuali accessi successivi da parte di altri lettori sono automaticamente accettati e sopravanzano quindi lo scrittore, che potrà dunque sperimentare fenomeni di starvation più o meno importanti. Gli spin lock rw permettono quindi di ottenere latenze medie minori se si ha una netta prevalenza di task lettori e non vi sono particolari vincoli temporali per quelli di tipo scrittore.

4.8.3 Semafori

A differenza dei precedenti, i semafori implementano un meccanismo nel quale i task in attesa che il lock sia rilasciato non mettono in atto il busy waiting ma vengono posti in stato di sleep e inseriti in una coda di attesa facente parte della struttura stessa del semaforo. Inoltre, essi hanno la possibilità di gestire la quantità di thread che possono accedere simultaneamente alla stes-

sa risorsa, attraverso un contatore interno che viene decrementato ad ogni nuova allocazione e decrementato in concomitanza di ciascun rilascio; se il contatore è posto inizialmente al valore 1 essi implementano effettivamente un accesso in mutua esclusione, altrimenti pongono solo un limite al numero di accessi simultanei, non garantendo ulteriori tipi di protezione.

La loro struttura ci permette di ricavarne alcune proprietà fondamentali:

- dato che i task in attesa vengono sospesi, i semafori sono partcolarmente adatti per tempi di attesa più elevati rispetto a quelli gestiti dagli spin locks.
- per contro, lo spazio aggiuntivo per la memorizzazione della coda di attesa e l'overhead dovuto a portare il task in stato sleep diventano fattori deleteri nella gestione di allocazioni brevi.
- la possibilità che i processi siano messi in stato di sleep consente l'uso dei semafori solo in contesto di processo. Inoltre, sempre per lo stesso motivo, non è possibile richiedere l'allocazione di un semaforo mentre si detiene già uno spinlock.

Anche dei semafori esiste la versione reader/writer, i cui principi di funzionamento coincidono con quelli visti per gli spin locks rw.

Altre strutture distinte ma simili nel comportamento ai semafori sono i *mutex*, i quali offrono solamente l'accesso in mutua esclusione ma possiedono un'interfaccia più semplice e migliori prestazioni, ottenute al prezzo di alcune limitazioni nel loro utilizzo: lo stesso processo che detiene il mutex deve provvedere al suo rilascio, non sono ammesse allocazioni ricorsive dello stesso mutex, un task che detiene un mutex non può terminare e non può essere acquisito da gestori di interruzioni o bottom half.

Infine, possiamo citare anche le *completion variables*, usate per sincronizzare kernel task in cui un processo deve attendere il completamento di un determinato lavoro da parte di un secondo task prima di poter proseguire. I task in attesa possono essere multipli e vengono gestiti tramite una coda FIFO [33].

4.8.4 Futex

I futexes [32] costituiscono un particolare meccanismo di lock a servizio dello spazio utente tale da non richiedere, in condizioni di acquisizione o rilascio non concorrente (ovvero in assenza di altri thread o task che in quel medesimo intervallo ne richiedono l'allocazione), l'intervento del kernel.

Essenzialmente si tratta di un indirizzo di memoria (ad esempio una variabile di lock a 32bit) in spazio utente; nel caso si verifichi un tentativo di accesso concorrente, tale variabile è marcata con un valore che indica la presenza di un processo in attesa, il quale viene successivamente inserito in una c̈oda futexc̈reata, attraverso una specifica chiamata a sistema, dal

kernel. Nel momento in cui il futex viene rilasciato, attraverso il valore della variabile il task vede se ci sono processi pendenti e, in caso affermativo, inoltra una nuova chiamata a sistema che provvede a risvegliare i task nella coda futex; una volta svuotata l'intera coda, nello spazio kernel non rimane alcuna traccia associata con il futex stesso.

I futex, quindi, si rivelano utili per gestire la sincronizzazione tra diversi processi in spazio utente in cui si stimano essere poco probabili le situazioni di contenzioso del lock, in quanto in tali condizioni l'intervento del kernel è ridotto al minimo.

4.8.5 Sequential locks

Introdotti a partire dal kernel 2.6, fornisce un ulteriore struttura per la gestione di dati condivisi tra processi lettori e scrittori. favorendo questi ultimi diversamente da quanto fatto dalle strutture di tipo reader/writer precedenti; l'intero meccanismo si fonda su una variabile, il *sequence counter*, sfruttata in modo diverso dai due tipi di processi.

I task scrittori, infatti, provvedono ad incrementarla (rendendola dispari) una volta ottenuto il lock e a decrementarla (e quindi facendola tornare pari) appena prima del rilascio; i lettori invece non necessitano di acquisire il lock, ma prima e dopo ogni operazione di lettura controllano il valore del sequence counter e, se questo rimane pari in entrambe le occasioni, significa che non sono intercorse scritture e quindi il dato acquisito è consistente, altrimenti viene ripetuta l'operazione di lettura.

In questo modo i processi scrittori non sono influenzati dal comportamento dei lettori, i quali invece possono dover ripetere più volte la stessa operazione di lettura prima che essa risulti valida, incrementandone così in modo imprevedeibile i tempi di esecuzione.

4.8.6 Read-copy update

Il read-copy update (RCU) [15] è un meccanismo di sincronizzazione introdotto in Linux dal 2002, che permette ad operazioni di lettura e aggiornamento di essere eseguite in modo concorrente senza la necessità di blocchi o ripetizioni di accesso (come nei lock sequenziali), Il fulcro del funzionamento risiede nel fatto che gli scrittori mantengono copie multiple della struttura dei dati condivisa, assicurandosi che le copie più vecchie (quindi non aggiornate) siano mantenute fino a quando i precedenti lettori che vi erano acceduti non hanno completato l'acquisizione dei dati. L'intero meccanismo si basa su tre principali operazioni:

- **Publication**: attraverso la funzione `rcu_assign_pointer()` viene creato un puntatore alla struttura condivisa e viene garantito che qualsiasi modifica effettuata sia visibile ai lettori che effettueranno una subscription alla struttura.

- **Subscription**: usa la primitiva `rcu_dereference()` per sottoscrivere la struttura creata nella precedente operazione, e questo viene realizzato passando un puntatore a tale struttura. Per garantire la corretta deferenziazione del puntatore, tutta la procedura è racchiusa tra le operazioni di `rcu_read_lock()` e `rcu_read_unlock()` e viene definita *RCU read-side critical section*.

- **Retraction**: l'obiettivo di questa fase si attua nel momento in cui i dati devono essere modificati; esso deve assicurarsi che il riferimento a quelli vecchi sia mantenuto solamente dai lettori che già avevano operato la subscription. La primitiva `synchronize_rcu()` attende quindi il completamento delle operazioni di lettura di questi reader (mentre le nuove subscription puntano alla nuova copia aggiornata). Una volta terminate, i vecchi dati non hanno più ragione di esistere e il sistema ne può recuperare lo spazio occupato.

L'assenza di blocchi o ripetizioni di accesso rende al RCU un ottimo candidato per garantire una maggiore prevedibilità dei tempi di aceesoo e modifica delle strutture condivise. La Figura 4.3 mette in luce il crescente utilizzo del meccanismo di RCU [30] nel kernel Linux a partire dalla sua introduzione nel 2002.

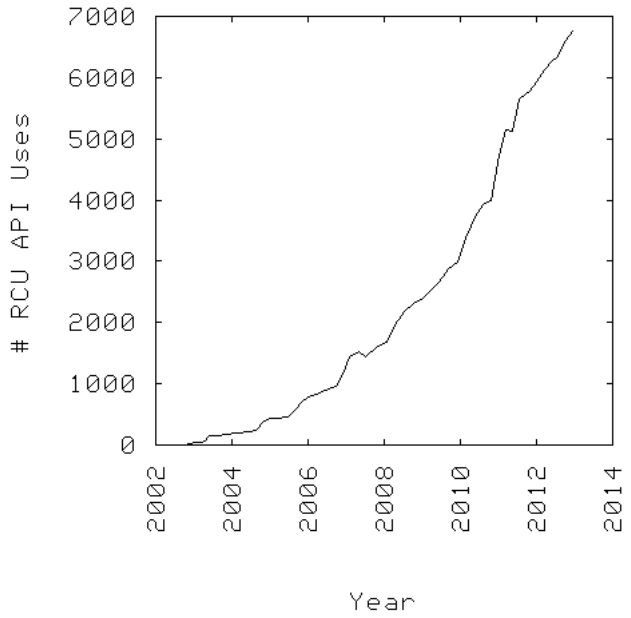


Figura 4.3: Andamento delle occorrenze di RCU nel kernel Linux

4.9 Gestione della memoria

Nella prima parte del capitolo abbiamo visto che uno dei principali compiti del kernel è quello di distribuire le risorse di sistema tra i vari task presenti. In questa sezione ci concentreremo sulla gestione della memoria di sistema, evidenziando le funzionalità fornite dal kernel per manipolarla e per renderne l'allocazione la più rapida ed efficace possibile.

4.9.1 Suddivisione della memoria

Il kernel gestisce la memoria fisica utilizzando come unità di misura base la pagina, ossia una porzione di memoria la cui dimensione è dipendente dall'architettura del sistema (generalmente pari a 4Kb per cpu a 32bit e 8Kb per quelle a 64bit). Questo perchè molti meccanismi hardware di gestione della memoria, come l'MMU (che si occupa della traduzione tra indirizzi virtuali e indirizzi fisici di memoria), lavorano con una granularità minima pari appunto alla dimensione di una pagina. Ciascuna pagina viene descritta dal kernel tramite la struttura `struct_page`, definita in linux/mm_types.h, di cui riportiamo la descrizione dei campi più importanti:

- **flags**:descrive lo stato della pagina, dichiarando ad esempio se per essa è attualmente disabilitato lo swap su disco (*page locked*). I valori ammissibili per questo campo sono contenuti in linux/page-flags.h.

- **_count**: tiene traccia dei riferimenti attivi alla pagina; quando raggiunge un valore negativo, la pagina è considerata libera e disponibile per nuove allocazioni.

- **virtual**: contiene l'indirizzo virtuale corrispondente alla pagina fisica a cui la `struct_page` fa riferimento; nel caso la pagina abbia subito uno swap su disco, il campo avrà valore `NULL`.

Il kernel, grazie a questa struttura, tiene traccia di tutte le pagine fisiche del sistema, definendo se siano o meno disponibili per nuove allocazioni, a quale processo siano attualmente allocate, e così via.

Il kernel Linux, inoltre, suddivide la memoria in partizioni dette *zone* per permettere al sistema di gestire alcune limitazioni hardware; sono presenti ad esempio la `ZONE_DMA`, riservata alle pagine sfruttate dal meccanismo di DMA dei dispositivi a 16bit (la zona è limitata ai primi 16MB), la `ZONE_NORMAL`, che contiene le pagine costantemente mappate dal sistema (ossia quelle in cui il campo `virtual` ha sempre un valore valido) e la `ZONE_HIGHMEM`, la quale contiene le pagine che possono subire il processo di swapping (generalmente corrisponde, nei sistemi x86, alla parte eccedente gli 896MB).

Le varie zone non sono comunque sempre presenti; nei sistemi a 64bit, che possono mappare ampie porzioni di ram fisica rispetto a quelli a 32

bit (limitati a soli 4GB), la `ZONE_HIGHMEM` non viene utilizzata, almeno nei sistemi embedded o di largo consumo.

4.9.2 Manipolazione della memoria

Le funzionalità rese disponibiii dal kernel per la manipolazione della memoria si possono dividere in due categorie principali:

- **Granularità di pagina**: consentono di creare o liberare spazi di memoria contigui pari a multipli della dimensione di una pagina. Sono definite all'interno dell'header /linux/gfp.h e, tra i parametri che esse richiedono, deve essere evidenziato il ruolo di `flags`, di tipo `gfp_t`, il quale definisce le modalità con cui l'allocazione deve essere effettuata; alcuni valori utilizzabili sono:
  - `GFP_ATOMIC`: l'allocazione ha una priorità elevata e non le è permesso alla funzione che la esegue di porsi in stato di sleep; per questo motivo, è utilizzata nell'implementazione di gestori di interruzioni o mentre un processo kernel detiene uno spinlock. Dato che non può interrompersi, le probabilità di eseguirla con successo sono minori rispetto alle altre chiamate, specialmente in condizioni di scarsa memoria disponibile.
  - `GFP_KERNEL`: definisce un'allocazione standard che può essere posta in stato di sleep. Generalmente è impiegata in contesto di processo, dove è consentito ad un task di poter essere bloccato.
  - `GFP_USER`: simile alla precedente ma utilizzato esplicitamente per allocare memoria a processi in spazio utente.
- **Granularità di byte**: quando si desidera ottenere porzioni di memoria inferiori alla dimensione di una pagina, il kernel rende disponibili un paio di funzioni, definite in linux/slab.h:
  - `void * kmalloc(size_t size, gfp_t flags)`: la funzione ritorna il puntatore ad una regione di memoria contigua di dimensione almeno pari a `size` byte in caso di successo, o altrimenti restituisce il valore `NULL`
  - `vmalloc()`: opera come `kmalloc` con l'unica eccezione di restituire una regione di memoria in cui i soli indirizzi virtuali sono contigui, mentre questo non vale necessariamente per quelli fisici.

  Anche per queste funzioni valgono le stesse proprietà del parametro `flags` precedentemente evidenziate. Malgrado la contiguità degli indirizzi fisici sia necessaria solo in particolari circostanze, la maggior parte del codice kernel utilizza la prima delle due per questioni di efficienza; le pagine ottennute con la `vmalloc()`, infatti, necessitano di

essere mappate ciascuna con una propria pagina e questo provoca una più frequente modifica nei dati presenti nel TLB (translation lookaside buffer, una cache hardware usata in molte architetture per memorizzare le mappature tra indirizzi virtuali e indirizzi fisici più usate e migliorare così i tempi di accesso medi).

4.9.3 SLAB, SLOB e SLUB

Il tempo necessario a completare le operazioni di allocazione della memoria può essere fortemente influenzato sia dalla quantità di memoria ancora disponibile, sia dalla collocazione stessa delle pagine fisiche libere all'interno di essa (le funzioni a granularità di pagina presentano maggiore probabilità di fallire se le pagine disponibili sono frammentate).

Per arginare queste problematiche, il kernel linux mette a disposizione un meccanismo chiamato *slab allocator*, che opera come una sorta di cache per strutture di dati generiche.

La sua realizzazione si è basata sulle seguenti premesse:

- Strutture dati usate frequentemente tendono ad essere allocate e rilasciate spesso, quindi conviene memorizzarle in cache per garantire un throughput migliore.

- Allocazioni e deallocazioni frequenti comportano la frammentazione della memoria, e una cache con una struttura particolare può contenere questo fenomeno.

- Il meccanismo deve essere indipendente dalla dimensione degli oggetti o delel pagine, in modo da realizzare una gestione ottimale.

- deve poter consentire un'allocazione locale per ciascuna cpu, in modo da eliminare la necessità di meccanismi di lock per gestire l'accesso alla memoria in sistemi multicore.

Lo slab divide oggetti differenti in gruppi chiamati *caches*, ciascuno contenente una specifica categorie di strutture; ad esempio, un cache può essere costituito da una lista di strutture `task_struct`. Ciascun cache è composto da uno o più *slab*, formati da una o più pagine fisicamente contigue; infine, all'interno degli slab sono presenti le strutture effettive di cui si vuole realizzare il meccanismo di cache. Ogni singolo cache dispone inoltre di un array per-cpu [34], organizzato secondo una logica LIFO e introdotto per incrementare la frequenza dei cache-hit e per ridurre la necessità degli spinlock per proteggere i dati condivisi nei sistemi SMP.

I slab possono essere completi, parziali o vuoti a seconda che le strutture in essi contenute siano rispettivamente tutte occupate, allocate solo in parte o completamente disponibili. Le richieste di oggetti da parte del kernel

vengono inizialmente servite dagli slab parziali e, quando questi sono esauriti, da quelli vuoti; ovviamente, se mancano anche questi ultimi ne vengono creati di nuovi (allungando solo in questo caso le tempistiche per completare l'allocazione).

Oltre allo SLAB, il kernel offre la possibilità di utilizzare altri due meccanismi:

- **SLOB**[35]: composto da un insieme di pagine inserite, nel tentativo di ridurre la frammentazione, in tre liste distinte: una per oggetti di dimensioni inferiori ai 256bytes, una per oggetti minori di 1024bytes e una terza per i rimanenti. Rispetto al precedente la sua implementazione richiede minori risorse ed è più semplice ed efficiente nell'uso dello spazio di memoria ma solo in sistemi con risorse limitate (e quindi maggiormente adatto per dispositivi embedded).

- **SLUB**[31]: simile allo SLAB (ne riutilizza la medesima interfaccia) ma progettato in modo da contenere l'overhead dovuto alle numerose strutture necessarie ad implementarlo.

4.9.4 Altri metodi di allocazione

Sempre all'interno del kernel, sono presenti due ulteriori modalità di gestione della memoria che possono influenzare la prevedibilità e l'efficienza nei tempi di risposta del sistema:

- **Gestione della memoria alta**: come già visto, le pagine presenti in questo tipo di memoria non sono mappate in modo permanente nello spazio di indirizzi del kernel; perchè il kernel possa utilizzarle, tale mappatura può essere realizzata ,una volta che le pagine sono allocate, tramite due distinte funzioni definite in linux/hghmem.h:

  - `static inline void *kmap(struct page *page)`: permette di creare una mappatura permanente della pagina. Ha lo svantaggio di essere uuna operazione bloccante e quindi utiizzabile solo in contesto di processo; inoltre, il numero di mappature permanenti è limitato, quindi le pagine in memoria alta dovrebbero essere rilasciate una volta terminatone l'uso.
  - `static inline void *kmap_atomic(struct page *page)`: viene utilizzata in contesti nei quali non è permesso lo stato di sleep, ed essa sfrutta un insieme di mappature temporanee che il kernel riserva appositamente a questo scopo. La funzione disabilita temporaneamente la preemption poichè tali mappature sono univoche per ciascun processore, e quindi l'interruzione di un task da parte di un secondo sullo stesso processore potrebbe comportare la perdita delle mappature precedentemente salvate.

- **Utilizzo di variabili per-cpu**: l'uso estensivo di variabili per cpu, generalmente realizzate con array nei quali ogni posizione è accessibile da una sola e predeterminata cpu del sistema, consente di eliminare la necessità di meccanismi di lock per proteggere l'accesso alle variabili e riduce la frequenza di invalidazioni della cache dei singoli processori (se il processore dovesse manipolare dati presenti all'interno della cache di altri processori, quest'ultima sarebbe resa non valida con maggiore frequenza, e quindi il suo effetto di miglioramento del throughput verrebbe fortemente ridimensionato). Rimane però da gestire l'interrompibilità, che determina in questo ambito due problematiche:

  - se il codice subisce preemption e la sue esecuzione viene spostata in un'altra cpu, i riferimenti ad eventuali variabili per-cpu non sono più validi (avendo mutato cpu, i riferimenti puntano ora ad una posizione errata dell'array).
  - se un secondo processo interrompe il precedente, può accedere alla medesima variabile in modo concorrente.

  In questo caso, è sufficiente disabilitare la preemption durante l'accesso a tali variabili.

  L'utilizzo di questo tipo di dati si è diffuso a tal punto, psecialmente in sistemi con un elevato numero di cpu, che Linux ha inserito dalla versione 2.6 un'apposita interfaccia chiamata *percpu* dedicata alla creazione e manipolazione delle variabiii per-cpu.

4.9.5 Gestione della memoria in spazio utente

Lo spazio di indirizzamento riservato a ciascun processo in spazio utente consiste generalmente in un intervallo contiguo di indirizzi virtuali, e ogni intervallo è specifico per il determinato processo; in altre parole, due processi distinti possono contenere dati diversi nel medesimo indirizzo virtuale visto da ciascuno di essi. Fanno eccezione i soli thread, che come abbiamo già visto possono condividere uno spazio di indirizzamento comune.

I processi possono accedere solamente allo spazio che viene loro riservato, indicato anche col nome di *memory areas*, ed eventuali estensioni o riduzioni sono possibili solo richiedendo l'intervento del kernel con opportune chiamate a sistema.

Il kernel tiene traccia delle caratteristiche dello spazio di indirizzamento di un processo, come l'estensione, i permessi, eventuali pagine con lock, ecc, attraverso un *descrittore di memoria*, di tipo `struct mm_struct` ,definito in linux/mm_types.h.

Una peculiarità interessante dei descrittori di memoria risiede nella capacità di memorizzare i riferimenti a tutte le aree di memoria che compongono lo spazio di indirizzamento di un processo in due distinte strutture:

- **mmap**: conserva le varie aree di memoria in una lista, e risulta quindi utile nel caso si debbano compiere operazioni che prevedono l'attraversamento sequenziale di tutti gli elementi.

- **mm_rb**: le aree vengono strutturate in un albero rosso-nero, ottimo per migliorare le prestazioni di funzioni di ricerca.

Quindi, a fronte di una maggiore complessità della `mm_struct` e di un incremento nello spazio occupato, si ottengono operazioni di accesso e gestione della memoria più efficienti.

Inoltre, come nel caso delle ricerche effettuate attraverso la funzione `find_vma()`, vengono impiegati anche meccanismi di caching dei risultati ottenuti, in modo da poter sfruttare il principio di località (ossia il fatto che dopo un'operazione nella stessa area di memoria sia probabille che le seguenti operazioni facciano ancora riferimento alla medesima area) e ottenere così un miglioramento delle prestazioni medie grazie ad un hit-rate dei dati in cache del 30-40%.

4.10 I/O scheduler e caching del disco

Un ultimo settore da considerare è la gestione dell'accesso ai dati nelle periferiche di sistema e, in modo particolare, ai dischi fissi.

4.10.1 Il virtual filesystem

Il kernel include uno strato software chiamato *virtual filesystem (VFS)* che funge da interfaccia tra i file di sistema e i programmi in spazio utente. Il suo funzionamento può essere evidenziato dallaFigura 4.4: l'applicazione in spazio utente effettua uan chiamata alla funzione `write()` che passa attraverso l'interfaccia fornita da VFS, la quale si occupa di richiamare la corretta funzione di `write()` specifica per il filesystem presente sul dispositivo. In questo modo, l'operazione in spazio utente può essere effettuata su qualsiasi dispositivo che fornisca un filesystem comprendente l'implementazione della corrispondente chiamata; il sistema privilegia quindi una maggiore adattabilità a fronte di un aumento degli strati software da impiegare per realizzare l'operazione.

Il VFS comprende vari oggetti base per rappresentare i vari modelli di gestione dei file, tra cui i principali sono:

- **superblock**: permette di descrivere la struttura di uno specifico filesystem.

- **inode**: rappresenta uno specifico file.

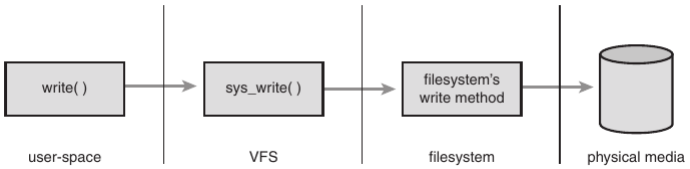


Figura 4.4: Strati software presenti tra una funzione in spazio utente e il dispositivo fisico di archiviazione dati

- **dentry**: abbinato ad una singola componente di un percorso; ad esempio, il percorso `a/b/c/d.txt` viene scomposto in quattro dentri, rispettivamente `a, b, c` e `d.txt`.

- **file**: a differenza dell'inode, questo oggetto rappresenta un file aperto, ovvero l'associazione tra un file e il processo che attualmente lo stà utilizzando.

Per gli oggetti di tipo dentry in particolare il kernel implementa anche un meccanismo di caching, poichè l'accesso ai file (e quindi l'utilizzo dei percorsi per raggiungerli) usualmente presenta sia aspetti di località temporale che spaziale, e quindi viene evitato di ricostruire ad ogni accesso l'intero percorso, operazione che risulta dispendiosa.

4.10.2 Scheduler I/O per dispositivi a blocchi

I dispositivi a blocchi sono le periferiche come hard disk, dischi blu-ray o memorie flash, nelle quali l'accesso ai dati avviene in modo random e non strettamente sequenziale. Gli sviluppatori del kernel hanno previsto un apposito strato software, il *block I/O layer*, per il loro controllo, in quanto ottimizzarne la gestione significava incidere notevolmente sulle prestazioni generali dell'intero sistema.

A livello hardware, la più piccola unità indirizzabile di un dispositivo a blocchi è il *settore*, la cui dimensione varia generalmente dai 512 byte crescendo via via per potenze di due. Per unificarne la gestione il kernel introduce la struttura del blocco, il quale deve rispettare alcuni vincoli: avere una dimensione pari ad uuna potenza di due e comunque multipla rispetto a quella minima di un settore, e inoltre non può superare la grandezza di una pagina.

I dati, per essere manipolati, vanno prima caricati in memoria, dove ciascun blocco è associato ad una nuova struttura, il *buffer*; dati i vincoli prima imposti, una pagina di memoria può dunque contenere uno o più buffer simultaneamente. Ogni buffer è abbinato ad una variabile di tipo `buffer_head` (linux/buffer_head.h) il cui compito principale è conservare le

corrispondenze tra i blocchi del disco e la posizione in memoria fisica delle loro copie.

Il problema principale di questi dispositivi (fortemente ridimensionato con l'arrivo delle memorie flash o dei solid state drive, nei quali si ha anche a livello hardware un reale accesso casuale) sono le operazioni di accesso ai vari blocchi del disco: ciascuna di esse, detta *seek*, comporta lo spostamento meccanico di una testina di lettura/scrittura nell'esatta posizione del blocco desiderato, e questo comporta tempi di attesa di svariati millisecondi. Di conseguenza, tutte le richieste di lettura e scrittura dati provenienti dal sistema (registrate in code specifiche) devono essere riorganizzate per minimizzare il numero di operazioni di seek necessarie a servirle; per fare ciò, il kernel implementa algoritmi di fusione e di riordino delle varie richieste attraverso uno dei suoi sottosistemi, l'*I/O scheduler*. Naturalmente, questo tipo di ottimizzazione ancora una volta privilegia un miglioramento nell'utilizzo della risorsa ma incide negativamente sulla prevedibilità dei tempi con i quali una singola richiesta di lettura o scrittura viene effettuata.

Il funzionamento dello scheduler si attua in due fasi:

- **merging**: se si hanno più richieste riguardano blocchi fisicamente adiacenti, queste vengono unificate in una sola richiesta, riducendo l'overhead e il numero di seek necessari.

- **sorting**: la coda di richieste viene ordinata in modo tale da minimizzare l'entità dello spostamento delle operazioni di seek (il cui tempo di esecuzione dipende dallo spazio che deve essere percorso per posizionare la testina in corrispondenza del cilindro contenente i blocchi desiderati)

Il kernel Linux mette a disposizione varie tipologie di schedulatori I/O, che andremo brevemente ad illustrare:

- **Linux Elevator**: quando una nuova richiesta viene aggiunta alla coda, viene effettuato un primo controllo per individuare eventuali candidati per una operazione di merging e, se presenti, la richiesta viene inserita rispettivamente prima o dopo di questi (a seconda che i blocchi relativi alla nuova richiesta precedano o seguano nell'ordine quelli delle richieste già accodate). In caso contrario, viene ricercata una posizione che minimizzi l'entità degli spostamenti e, se anche questa non esiste, il nuovo elemento viene posizionato in fondo alla coda.

  La coda viene servita in modalità FIFO e, per evitare il fenomeno della starvation, non si consente l'inserimento di nuove richieste davanti a quelle presenti in coda da più di un intervallo di tempo predefinito; il meccanismo non è però efficiente e, malgrado migliori la latenza, non riesce ad annullare totalmente l'impatto della starvation. L'algoritmo, seppur ancora presente nel kernel, era usato nella versione 2.4 ed è stato sostituito dai successivi a partire dalla 2.6.

- **Deadline I/O scheduler**: inglobato nel kernel con l'obiettivo di risolvere definitivamente il problema della starvation. In modo particolare si osserva che, mentre le operazioni di scrittura richieste da un'applicazione sono generalmente effettuate in modo asicrono ad essa (non sono bloccanti), quelle di lettura invece agiscono nel modo opposto; da questa constatazione, segue che la responsività dell'applicazione è maggiormente influenzata dalla latenze delle operazioni di lettura.

  Inoltre, le letture sono generalmente dipendenti l'una dall'altra (ovvero la successiva non inizia se la precedente non è completata) e quindi le latenze di letture consecutive vanno generalmente a sommarsi. Obiettivo dell'algoritmo è quindi quello di minimizzare la starvation delle letture, a costo di un minore throughput nell'utilizzazione della periferica. Vediamo per punti come opera il Deadline I/O scheduler:

  - ad ogni nuova richiesta è associata una scadenza, pari a 500ms nel caso delle letture e 5s per le scritture.
  - il sistema mantiene tre code distinte: una ordinata simile a quella dell'algoritmo precedente, e altre due nelle quali sono inserite rispettivamente le operazioni di lettura e quelle di scrittura, gestite secondo logica FIFO(quindi gli elementi rispettano l'ordine di arrivo).
  - le tre code vengono utilizzate per determinare l'effettivo ordine con cui le richieste sono servite: usualmente, queste vengono prelevate dalla coda ordinata ma, nel caso in cui si verifichi nelle altre due la scadenza dell'intervallo di attesa, allora i nuovi elementi sono acquisiti da queste altre due code.

  Pur non potendo garantire il rispetto di latenze di risposta in modo assoluto, generalmente riesce a servire le richieste entro le scadenze fissate, favorendo le letture grazie all'intervallo di attesa più breve ad esse abbinato. Il punto debole dell'algoritmo si esplica nelle condizioni nelle quali sono presenti molte scritture intervallate da poche letture; favorendo queste ultime, infatti, ad ogni lettura avremo un seek per spostare la testina nei blocchi richiesti e un secondo per proseguire con la sequenza di scritture, comportando così un throughput minore.

- **Anticipatory I/O scheduler**: ideale per i sistemi server, mira a ottenere un throughput migliore del precedente pur mantenendo una eccellente latenza nelle letture. L'unica differenza dal precedente risiede nel fatto che, dopo aver servito una lettura, l'algoritmo non torna subito alle operazioni di scrittura ma attende per un predefinito lasso di tempo (6ms di default) il possibile arrivo di nuove letture in aree adiacenti; in caso affermativo, queste sono servite subito, risparmiando

così la coppia di operazioni di seek che sarebbe altrimenti stata necessaria per tornare alle operazioni di scrittura e successivamente servire la seconda operazione di lettura.

Come si può evincere, il miglioramento dato dall'algoritmo si ha solo se il tempo trascorso in attesa di nuove letture è inferiore a quello risparmiato eliminando le coppie di seek non più necessarie; per realizzare ciò, l'algoritmo cerca di prevedere il comportamento di ogni processo registrandone un insieme di dati statistici che gli permettano così di stabilire con una maggiore percentuale di riuscita gli istanti in cui conviene o meno attivare il periodo di attesa.

- **CFQ I/O scheduler**: mantiene una coda di richieste per ciascun processo presente nel sistema; ciascuna coda subisce i processi di merging e sorting e le varie code presenti sono servite in modo round-robin, ovvero servendo una quantità predeterminata di richieste per ciascun processo. L'algoritmo consente di realizzare un sevizio equo a livello di processo e, benchè sia stato ideato specificamente per applicazioni di tipo multimediale, offre buone prestazioni in un ampio insieme di scenari; attualmente costituisce l'I/O scheduler di default.

- **Noop I/O scheduler**: si limita a realizzare solamente l'operazione di merging, senza attuare alcun ordinamento. È pensato per dispositivi quali le memorie flash o gli SSD, in quanto il loro funzionamento non prevede parti meccaniche in movimento e dunque le operazioni di seek hanno un costo praticamente nullo.

4.10.3 Il page cache

Il kernel Linux include anche una cache in RAM per il disco, la *page cache*, con lo scopo di salvare in memoria i dati acceduti più frequentemente e, sfruttando il principio di località temporale, minimizzare il numero di accessi allo stesso disco, generalmente caratterizzati da tempi di esecuzione di vari ordini superiori a quelli della RAM.

Il page cache consiste in un insieme di pagine fisiche il cui contenuto corrisponde a quello di uno o più blocchi di un disco; questo insieme può variare la sua cardinalità a seconda del quantitativo di memoria ancora disponibile in un dato istante. In questo modo, le operazioni di lettura sono realizzate in modo da verificare primariamente la presenza dei dati cercati in cache e, in caso negativo, viene schedulata una nuova operazione reale di accesso al dispositivo I/O e i nuovi dati letti vengono a loro volta spostati all'interno della cache.

Per quanto riguarda invece le operazioni di scrittura, queste possono essere realizzate sfruttando tre strategie differenti:

- **No-write**: la scrittura viene eseguita direattamente su disco, marcando come non più valida la corrispondente posizione in cache. Viene usata raramente in quanto sfrutta debolmente la cache e la funzione di invalidazione è temporalmente costosa.

- **Write-through**:vengono aggiornati sia la cache che i dati su disco; è semplice e non necessita di verificare o modificare la validità della cache.

- **Write-back**: è la metodologia presente nel kernel Linux e consiste nel scrivere i dati solamente in cache, marcando la pagina interessata come *dirty* e inserendola in una *dirty list*. Periodicamente, le pagine presenti in questa lista vengono scritte anche su disco (operazione di *write-back*) e viene eliminata la marcatura. Presenta il vantaggio di raggruppare assieme più scritture eseguendole quindi con efficienza maggiore, ma risulta più complesso da realizzare e gestire.

La gestione della cache si completa anche di una procedura (*cache eviction*) che si prende carico di decidere quali dati rimuovere dalla cache per fare posto ad informazioni più recenti o per liberare spazio di memoria necessario a task più prioritari. Le sole pagine ad essere incluse in questo processo sono quelle non marcate come dirty; la strategia ottimale di rimpiazzo però necessiterebbe di conoscere anche l'andamento futuro delle varie attività del sistema, quindi il kernel cerca di approssimarne il comportamento attraverso l'implementazione del secondo di questi due possibili approcci:

- **Leas recently used (LRU)**: il kernel tiene traccia degli istanti di accesso alle varie pagine della cache e, nel caso si renda necessaria la loro eliminazione, inizia scegliendo quelle con istante di accesso più lontano nel tempo. L'algoritmo offre generalmente buone prestazioni eccetto nei casi in cui molti dati siano acceduti una sola volta(potrebbero portare all'eliminazione di pagine ad accesso più lontano nel tempo ma verso le quali sono richiesti accesi multipli).

- **Strategia delle due liste**: vengono mantenute una lista attiva di pagine non rimpiazzabili ed una inattiva di pagine candidate ad essere sostituite; le pagine sono poste nella lista attiva nel caso esse siano accedute mentre si trovano già in quella inattiva. Le due liste sono di tipo FIFO e vengono mantenute bilanciate: se la lista di quelle attive diviene troppo lunga rispetto all'altra, un certo numero di elementi alla sua testa vengono posti in coda alla lista inattiva; viene così risolto il problema dell'algoritmmo precedente.

Per quanto riguarda la gestione del write-back, essa viene affidata ad un insieme di thread del kernel indicati col nome di *flusher threads*, i quali si attivano al realizzarsi di una delle seguenti condizioni:

- la quantità di memoria libera del sistema scende sotto una data soglia prestabilita e quindi con l'operazione di write-back si creano delle nuove pagine "puliteë quindi rimpiazzabili o eliminabili.
- quando le pagine rimangono dirty per un intervallo di tempo eccessivamente lungo, evitando così che lo restino indefinitamente senza aggiornare mai i dati su disco.
- quando un processo in spazio utente invoca le chiamate `sync()` o `fsync()` con cui richede esplicitamente al kernel di eseguire l'operazione di write-back.

Il comportamento dei flusher thread può essere modificato per adattarlo alle esigenze di risparmio energetico dei dispositivi con alimentazione a batteria, nei quali si cerca generalmente di mantenere i dischi inattivi per intervalli il più lunghi possibile. Questo viene attivato attraverso la variabile `/proc/sys/vm/laptop_mode`; l'unico cambiamento ingenerato dalla modifica riguarda la seconda delle condizioni prima elencate, al verificarsi della quale, in questo caso, vengono effettuati i write-back su tutte le pagine dirty presenti. La modifica può essere fatta a runtime, adattandosi a seconda se l'alimentazione avviene da rete o da batteria.

Capitolo 5

Android

5.1 Sommario

All'interno del capitolo verrà fornita una descrizione generale del sistema operativo Android, approfondendone quelle caratteristiche, basate sulle funzionalità fornite dal kernel Linux o peculiari della struttura stessa secondo cui è stato progettato, che ne determinano un più forte adattamento ad operare in ambienti soft real-time su dispositivi embedded.

5.2 Storia

Android Inc.[36] è stata fondata nell'ottobre 2003 da Andy Rubin, Rich Miner, Nick Sears e Chris White], per lo sviluppo di quello che Rubin definí, «... dispositivi cellulari più consapevoli della posizione e delle preferenze del loro proprietario». Inizialmente la società operò in segreto, rivelando solo di progettare software per dispositivi mobili.

Il 17 agosto 2005 Google ha acquisito l'azienda, in vista del fatto che la società di Mountain View desiderava entrare nel mercato della telefonia mobile. È in questi anni che il team di Rubin comincia a sviluppare un sistema operativo per dispositivi mobili basato sul kernel Linux.

La presentazione ufficiale del robottino verde venne data il 5 novembre 2007 dalla neonata OHA(Open Handset Alliance), un consorzio di aziende del settore Hi Tech che include Google, produttori di smartphone come HTC e Samsung, operatori di telefonia mobile come Sprint Nextel e T-Mobile, e produttori di microprocessori come Qualcomm e Texas Instruments. Il primo dispositivo equipaggiato con Android lanciato sul mercato fu l'HTC Dream, il 22 Ottobre del 2008.

Dal 2008 gli aggiornamenti di Android per migliorarne le prestazioni e per eliminare i bug delle precedenti versioni, sono stati molti. La versione piú recente è la 4.2 JellyBean.

5.3 Struttura generale

Android è uno stack software completo open-source, rilasciato sotto licenza Apache; gli sviluppatori hanno quindi accesso all'intero codice sorgente, consentendone una più rapida portabiilità su nuovi componenti hardware, garantendo al contempo la possibilità di aggiungere codice proprietario senza necessariamente rilasciarne i sorgenti corrispettivi (cosa prevista invece per la licenza GPL v2 con la quale è rilasciato il kernel Linux).

Essendo progettato per adattarsi ai dispositivi mobile, molta attenzione è stata dedicata ai seguenti aspetti:

- riduzione dei consumi energetici, dato che malgrado i miglioramenti tecnologici apportati le batterie non permettono ancora autonomie prolungate.
- le ridotte dimensioni, che comportano minori quantità di memoria e cpu dalle prestazioni ridotte (rispetto ai corrispettivi modelli per desktop).
- portabilità, in quanto il sistema Android non pone alcun limite a specifiche, come dimensionee e risoluzione dello schermo o chipset integrati, dei dispositivi in cui viene installato.

Lo stack software [37]che compone Android altro non è che l'insieme di un kernel Linux e di un insieme di librerie in linguaggio C/C++ rese accessibili attraverso un framework sviluppato in Java che fornisce servizi per le applicazioni che andranno ad esservi eseguite. L'intero stack è formato dai componenti presenti in Figura 5.1 , che andremo brevemente ad ilustrare:

- **Kernel Linux**: fornisce i servizi di base quali i driver per le periferiche, la gestione dei processi e della memoria, la sicurezza, le comunicazioni di rete e la gestione dei consumi. Inoltre, agisce come uno strato che astrae le peculiarità dell'hardware sottostante per presentare un'interfaccia di interazione omogenea ai rimanenti strati software che compongono il sistema.
- **Librerie**: sono incluse varie librerie scritte in C/C++ tra le quali:
  - una versione riveduta delle libc, chiamata bionic;
  - una libreria multimediale per la decodifica di flussi audio e video;
  - il *Surface manager* per la gestione degli oggetti presentati sullo schermo dei dispositivo;
  - librerie SGL e OpenGL ES rispettivamente per la grafica 2D e 3D
  - SQLite per garantire il supporto nativo ai database;
  - SSL e WebKit per il supporto alla sicurezza di rete e ai browser

- **Ambiente di runtime**: è la parte che distingue Android dall'essere un semplice adattamento di Linux ai dispositivi mobili, e costituisce il motore che permette l'esecuzione delle applicazioni e, assieme alle *core libraries*, coastituisce la base del livello software superiore. Nello specifico, le core libraries forniscono la gran parte delle funzionalità presenti nelle librerie Java standard, mentre la *Dalvik VM* è una macchina virtuale basata su registri e ottimizzata per dispositivi embedded, capace di eseguire efficientemente istanze multiple di se stessa e basata sulle funzionalità del kernel per quanto riguarda la gestione dei thread e della memoria.
- **Application framework**: mette a disposizione le classi utilizzate per creare le applicazioni per Android, fornendo al contempo un'interfaccia per l'accesso all'hardware e la gestione dell'interfaccia utente e delle risorse necessarie ai processi.
- **Applicazioni**: raggruppa tutti i programmi, nativi o sviluppati da terze parti, che consentono all'utente finale di interagire con il dispositivo.

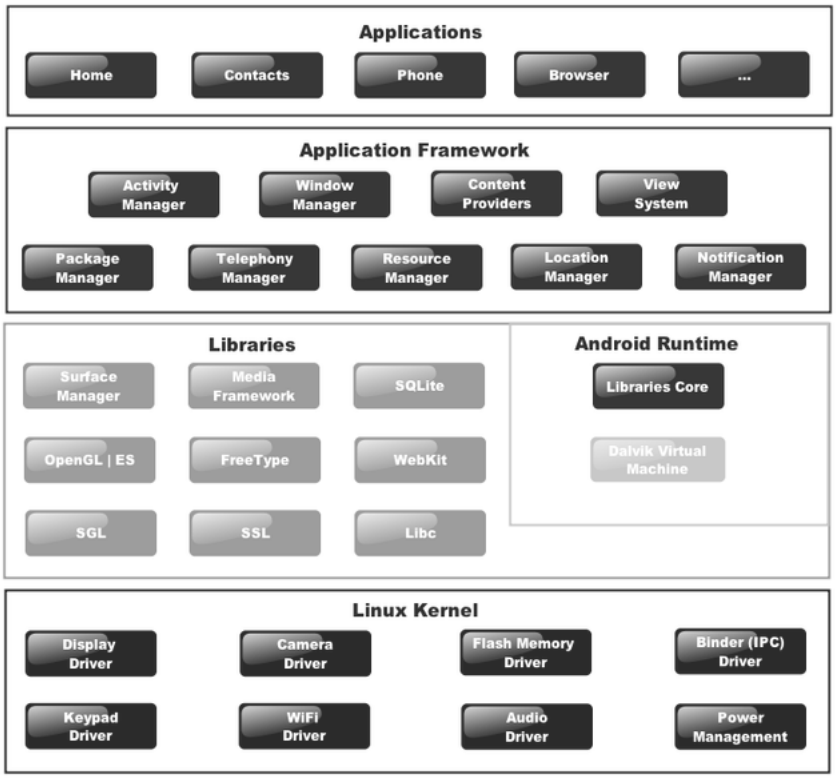


Figura 5.1: Il framework software di Android

5.4 Rapporto con il soft real-time

In questa sezione andremo ad analizzare più in dettaglio le componenti del sistema mirate a renderlo più adatto ad un ambiente di tipo soft real-time, realizzando risposte agli eventi esterni più deterministiche o minimizzando i tempi di reazione del sistema.

Nel contempo, va incluso il fatto che il sistema deve considerare anche i vincoli dati dall'ambiente di tipo embedded in cui opera e quindi non può trascurare elementi come la scarsità di memoria, la relativamente bassa potenza elaborativa delle cpu e il contenimento dei consumi energetici.

Nell'ordine saranno approfondite le seguenti componenti:

- Il funzionamento della Dalvik VM
- Il processo Zygote.
- il Garbage collector.
- Componenti delle applicazioni.
- Lo stack delle applicazioni.
- Processi e loro priorità.
- L'utilizzo dei thread.
- La gestione dei consumi (wakelock).
- Utilizzo di codice nativo.
- Programma di compatibilità.

5.4.1 Dalvik VM

Il linguaggio Java [16] si è sempre fregiato del fatto che, una volta scritto, il codice sarebbe stato eseguibile ovunque; questo grazie alla particolare piattaforma del linguaggio stesso, basata su una macchina virtuale che si occupa di astrarre le peculiarità dell'hardware sottostante presentando un ambiente runtime sempre uniforme. Questo però ha comportato la creazione di varie tipologie di VM, dedicate a vari contesti (desktop, server e mobile); in modo particolare l'ambiente mobile si è rivelato maggiormente frammentato a causa di configurazioni, profile e pacchetti supportati, costringendo a introdurre modifiche più o meno marcate alle applicazioni per supportare dispositivi differenti.

Anche Android si appoggia al linguaggio Java per la realizzazione delle applicazioni e del framework alla base di queste, ma ha scelto di sviluppare una propria macchina virtuale, la *Dalvik virtual machine (DVM)*, le cui caratteristiche le permettessero di adattarsi ai molteplici dispositivi in ambito

mobile e alle loro limitazioni hardware, fornendo al contempo un ambiente di esecuzione separato per ciascuna applicazione (*sandbox*) in modo che queste non interferissero tra loro o con la piattaforma Android. La JVM non era inoltre liberamente distribuibile in termini di licenza e quindi lo svluppo di un'alternativa open source e con licenze meno restrittive ha contribuito a fornire una piattaforma più appetibile per essere adottata su un'ampia varietà di dispositivi.

La DVM è stata progettata in modo da essere eseguita su cpu bassa potenza, con quantitativi limitati di ram (si sono mantenuti i 64MB come target minimo), in presenza di sistemi operativi senza funzionalità di swap e di dispositivi alimentati a batteria.

È inoltre progettata in modo da eseguire efficacemente più istanze di se stessa, ognuna delle quali va a costituire un processo separato in cui verrà eseguita una determinata applicazione, attraverso l'adozione del processo Zygote.

La DVM utilizza un bytecode differente da quello standard di Java, ottimizzato per avere la minima occupazione di memoria. Mentre Java crea un file per ciascuna classe dichiarata nel file sorgente, il formato .dex permette di ottenenere un unico file contenente più classi simultaneamente, come illustrato nella Figura 5.2.

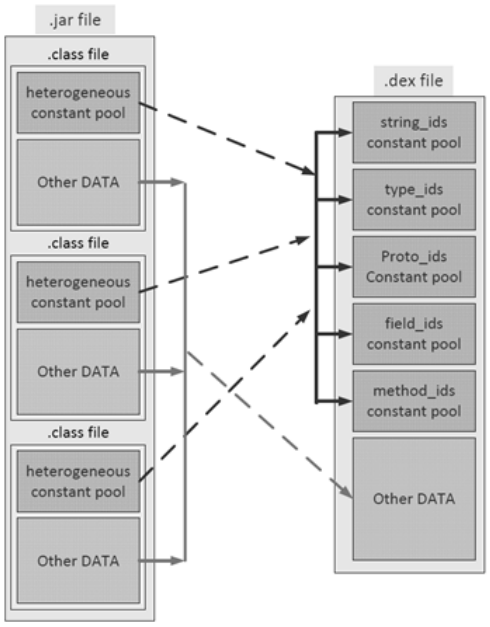


Figura 5.2: Conversione dal bytecode di Java a quello adottato dalla DVM

Il file .dex utilizza una suddivisione del file in porzioni specifiche, condivise e contenenti oggetti di un tipo predefinito, come ad esempio stringhe

	.jar non compresso	.jar compresso	.dex
Librerie di sistema	21445320 (100%)	10662048(50%)	10311972(48%)
Web browser	470312 (100%)	232065 (49%)	209248 (44%)
Alarm clock	119200 (100%)	61658 (52%)	53020 (44%)

Tabella 5.1: Dimensioni occupate dai vari formati di bytecode

di valore costante, tipi di variabili o classi, prototipi di metodi, ecc. Questo permette notevoli vantaggi dal punto di vista dello spazio occupato:

- Si evitano tutte le duplicazioni di oggetti, presenti anche in classi diverse. Ciascun elemento può quindi essere definito con un ID univoco e le sue componenti definite con riferimenti agli oggetti conservati nel file .dex; ad esempio, una variabile potrà essere definita tramite riferimenti alla stringa che ne definisce il nome, al tipo e alla classe di appartenenza.
- Il tipo predefinito delle varie porzioni del file rende inutile, ad esempio, dichiarare che una data costante è una stringa, poichè questo si ricava implicitamente dal tipo di sezione in cui è contenuta.

L'efficacia di un tale approccio [39] è stata dimostrata verificando che, in media, i file di tipo .dex arrivano ad occupare uno spazio minore addirittura delle versioni compresse (.jar) dei file compilati standard di Java, come evidenziato nella tabella 5.1 (le dimensioni sono date in byte):

Altra fondamentale differenza tra la JVM e le DVM è l'architettura di base; sono infatti adottati rispettivamente [38]:

- **Approccio basato su stack**: gli operandi di ciascuna istruzione sono contenuti in uno stack LIFO in memoria, nel quale gli operandi possono essere inseriti o prelevati tramite operazioni di push e pop. Il vantaggio risiede nel fatto che gli operandi sono indirizzati implicitamente attraverso lo stack e quindi la VM non deve forzatamente conoscerne gli indirizzi reali.
- **Approccio basato su registri**: in questo caso gli operandi sono conservati in specifici registri della cpu, e dunque ciascuna istruzione deve esplicitamente dichiarare anche le locazioni reali degli operandi che andrà ad utilizzare, divenendo quindi più lunga in termini di byte occupati. Una tale soluzione ha il vantaggio di eliminare tutto l'overhead legato alle operazioni di gestione dello stack, in particolar modo i numerosi accessi ala memoria, può usufruire, se l'hardware lo consente, di alcune tecniche di ottimizzazione (ad esempio conservando risultati intermedi in altri registri della cpu da utilizzare in istruzioni successive) e permette di avere istruzioni semanticamente più dense,

poichè contengono gli indirizzi dei registri in cui i dati da utilizzare sono effettivamente conservati.

Alcuni test comparativi vedono che in media l'architettura basata su registri richiede il 47% in meno di istruzioni e, malgrado siano il 25% più lunghe, richiedono caricamenti aggiuntivi di codice solo in poco più dell'1% dei casi, ottenendo dunque un miglioramento dei tempi di esecuzione che si attesta intorno al 32%.

Poiche la DVM deve operare in contesti che presentano cpu di potenza elaborativa relativamente bassa, la scelta dell'architettura basata su registri risulta vincente dal punto di vista prestazionale, mentre la maggiore occupazione in memoria dovuta alla lunghezza delle istruzioni è mitigata sia dal loro numero inferiore che dalla condivisione spinta operata tramite il formato .dex. Inoltre [38], la DVM esprime ciascuna istruzione con 16 bit (riservando campi di 4 bit per indicare il singolo registro), mentre il bytecode Java standard utilizza solamente 8 bit.

Le scelte operate dai progettisti di Android rivelano quindi che, per adattarsi ad ambienti di tipo soft real-time, si punta soprattutto sull'ottimizzazione spinta dei tempi di esecuzione piuttosto che implementando strategie di controllo viste nel capitolo tre.

5.4.2 Zygote

Come precedentemente ricordato, la creazione di nuove istanze della VM deve essere un processo rapido in quanto, oltre ad essere un'operazione frequente, influisce sui tempi del primo avvio di una applicazione; nell'ipotesi che vi sia un gran numero di librerie di base e delle relative heap condivise e utilizzate dalle varie VM, l'idea è quella di abilitarne la condivisione tramite un qualche meccanismo di controllo.

La soluzione è lo stesso processo Zygote (vedi Figura 5.3), creato al boot del sistema, il quale si occupa di inizializzare una DVM la quale precarica un insieme di classi base, generalmente a sola lettura e quindi ottimali per essere condivise (non vi è il problema di accessi in mutua esclusione). Una volta terminata l'inizializzazione, Zygote si pone in attesa e, alla richiesta di attivare una nuova applicazione, esegue un fork di se stesso, ponendo le classi di base come risorse condivise e quindi evitando di duplicarle e di dover trovare ulteriore spazio in memoria per caricarle, ottenendo così un più rapido startup dell'applicazione. Nel caso queste classi debbano essere scritte dall'applicazione, si va ad utilizzare il meccanismo di copy-on-write fornito dal kernel Linux sottostante, con vantaggi e svantaggi visti alla sezione 4.4.2.

Sempre allo scopo di ottenere migliori risultati sia sotto il profilo delle prestazioni che dell'occupazione di memoria, alcune delle librerie maggiormente condivise sono state riscritte da zero invece di essere semplicemente ereditate da quelle già presenti nel kernel Linux. Un esempio notevole è co-

stituito dalle libreire *bionic* [47], che vanno a sostituire le `glibc` del kernel Linux e presentano le seguenti peculiarità:

- Sono rilasciate sotto una licenza BSD e non GPL.
- Presentano una dimensione dimezzata rispetto alle `glibc`
- Inglobano una implementazione dei pthread (POSIX thread) più efficiente e alcune funzionalità aggiuntive specifiche per Android.
- per contro, non sono compatibili con le `glibc`, non supportano alcune proprietà come la gestione delle eccezioni del C++ e quindi il codice nativo deve essere ricompilato sempre utlizzando le bionic.

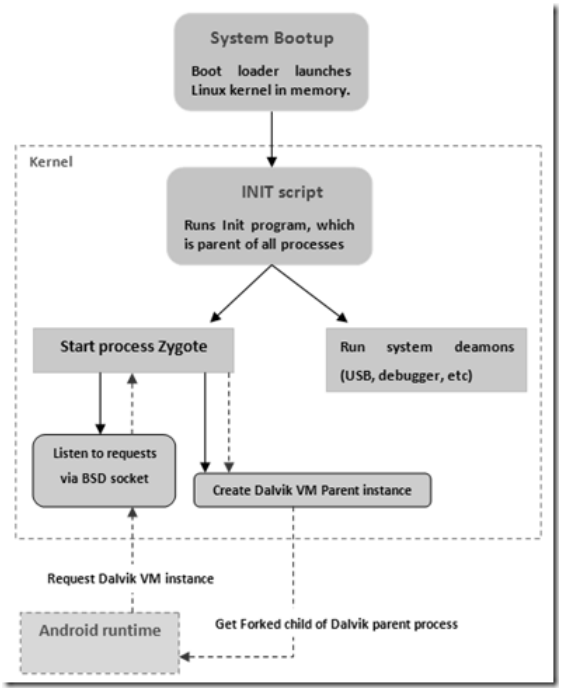


Figura 5.3: Il processo Zygote

5.4.3 Garbage collector

Un'ulteriore fonte di imprevedibilità nei tempi di risposta risiede nella presenza del *Garbage collector (GC)*, il processo che si occupa di riconoscere gli oggetti non più referenziati il cui spazio occupato in memoria può quindi essere liberato e reso disponibile per altri scopi, poichè i suoi istanti di attivazione e il tempo di esecuzione non sono determinabili a priori.

In Android, il GC [18] deve integrarsi con la modalità di condivisione della memoria tra i processi introdotta da Zygote, che come abbiamo visto

mantiene un unico spazio di memoria per gli oggetti condivisi, mentre alloca uno spazio privato per i dati dinamici di ciascun processo.

Il meccanismo con cui è strutturato il GC è il *mark-sweep*, distinto in due fasi:

- **mark**: in ciascun processo, vengono individuati gli oggetti immediatamente raggiungibili (ovvero i cui riferimenti sono presenti nello stack o nei registri) e viene costruito il grafo di tutti gli oggetti ricorsivamente raggiungibili da questi; tutti gli oggetti ritrovati costituiscono i cosiddetti *live-objects*, dotati quindi di almeno un riferimento valido e quindi ancora utilizzati dall'applicazione. Questa fase viene eseguita sia sulla memoria condivisa che su quella privata del processo. La marcatura avviene settando un array di bit mantenuto solamente per il tempo in cui il GC viene eseguito (per minimizzare l'occupazione di memoria).

- **sweep**: vengono eliminati gli oggetti privi di riferimenti e resa nuovamente disponibile la memoria da questi occupata, ma solo limitatamente alla porzione privata del processo.

In Android la parte condivisa è generalmente più grande rispetto a quella privata, quindi il tempo della fase di mark in essa è generalmente maggiore, e inoltre le operazioni di scrittura sono particolarmente onerose perchè è necessario richiamare il meccanismo di copy-on-write(sezione 4.4.2).Questo comporterebbe un tempo elevato per operare lo sweep anche su di essa, e dunque viene solamente interessata dalla prima fase per accertare la corretta raggiungibilità di tutti gli oggetti utilizzati.

Le versioni di Android precedenti la 2.3 [54] prevedevano un GC la cui attivazione andava a bloccare tutti i thread dell'applicazione e le cui fasi di mark e sweep interessavano l'intera memoria privata del processo; questo andava ad impattare sull'applicazione con ritardi spesso superiori ai 50-100ms e quindi inaccettabili in alcune attività con vincoli in tempo reale più stringenti e comunque percepibili dall'utente.

Successivamente, un deciso miglioramento nelle prestazioni temporali medie è stato ottenuto adottando una versione concorrente del GC, attuata in un thread separato che entra in funzione solo quando tutti gli altri thread dell'applicazione sono sospesi; in questo modo, si è contenuto il ritardo medio dovuto all'intervento del GC sotto i 5ms.

Nonostante i miglioramenti apportati, le stesse tecniche di programmazione possono incidere molto sulla frequenza con cui il GC deve intervenire [48]; non allocare oggetti utilizzati per intervalli molto brevi (come ad esempio quelli creati all'interno di un ciclo for) o utilizzare dove possibile tipi primitivi di variabili al posto di variabili oggetto (un array di interi contribuirà alla creazione di un minor numero di oggetti di un array di `Integer`, tenendo presente la contropartita che, se di dimensione elevata, il primo potrebbe andare ad occupare l'intero stack abbinato al processo, mentre nel

secondo caso lo stack vedrebbe aggiunto un solo riferimento all'intero vettore, salvato nella heap), consentirebbe di avere un minor numero di oggetti in memoria e dunque una riduzione media del numero di interventi del GC in una data unità di tempo.

5.4.4 Componenti delle applicazioni

Una volta che un'applicazione viene installata in un sistema Android, essa opera sempre in un ambiente ben definito (*sandbox*), caratterizzato dalle seguenti proprietà:

- A ciascuna applicazione viene abbinato, grazie al kernel Linux, un UID (id utente), utilizzato esclusivamente dal sistema per inizializzare i permessi relativi ai file dell'applicazione stessa, in modo solamente quel determinato UID possa accedervi.
- Ogni processo viene eseguito in una istanza separata della DVM, in modo da non interferire con le altre applicazioni presenti.
- Come comportamento di default, Android associa ad ogni applicazione un processo Linux separato, attivandolo nel momento in cui viene eseguita una qualsiasi componente della stessa applicazione e terminandolo nel momento in cui questo non è più necessario o il sistema deve liberare memoria e risorse a favore di altri task.

In questo modo viene realizzato il principio del minimo privilegio, ovvero ad ogni applicazione è proibito l'accesso a parti del sistema o ad altre applicazioni per le quali non abbia ricevuto una autorizzazione esplicita.

Nonostante ciò, sono messe a disposizione due diverse metodologie per rendere possibile la condivisione di risorse:

- due applicazioni possono condividere lo stesso UID e quindi accedere liberamente ai rispettivi file; inoltre, esse possono essere eseguite nel medesimo processo, utilizzando quindi una sola istanza della DVM per entrambe.
- grazie ad un sistema di comunicazione interprocesso realizzato tramite messaggi (*Intent*), le applicazioni possono richiedere al sistema operativo sottostante la possibilità di accedere a dati del sistema condivisi, come ad esempio lo spazio di storage delle schede SD; in questo caso, però, la necessità di tali permessi va dichiarata nei file `AndroidManifest.xml`, che costituisce parte integrande del programma stesso.

Le componenti di un'applicazione [40] sono i blocchi essenziali con cui i programmi Android vengono composti; ciascuna di esse costituisce un elemento indipendente e con ruoli specifici che concorre a definire l comportamento dell'intera applcazione.

Il sistema Android fornisce quattro tipi differenti di componenti:

- **Activities**: rappresenta una componente associata ad una schermata dell'interfaccia utente. Una applicazione può essere formata da una molteplicità di esse, ma operano in modo indipendente l'una dall'altra e solo una alla volta può assurgere al ruolo di componente attiva e interagire con l'utente.

- **Services**: componente che lavora in background, generalmente utilizzato per effettuare operazioni lunghe o che prevedono elaborazione remota. Non possiede alcuna interfaccia utente e la sua esecuzione non va a bloccare l'esecuzione dell'activity corrente.

- **Content providers**: è un oggetto che permette la condivisione di dati tra le applicazioni, indipendentemente da come e dove sono conservate. Ciascuna applicazione, a seconda dei permessi di accesso, può leggere o modificare i dati condivisi.

- **Broadcast receivers**: è un componente che reagisce a messaggi originati dal sistema (schermo spento, batteria con bassa carica, ecc) o da altre applicazioni (se queste inizializzano un proprio broadcast receiver).Anch'esso non possiede un'interfaccia ma può avvisare l'utente del verificarsi di dati eventi attraverso notifiche, e il suo compito principale è di inizializzare un qualche servizio che realizzi un'azione in risposta al realizzarsi dell'evento.

Diversamente dai normali programmi Java o C, un'applicazione di Android può avere più punti di ingresso, ciascuno in corrispondenza di una delle componenti che la costituisce. Questa caratteristica permette di implementare in Android un comportamento del tutto peculiare, per cui le componenti possono essere attivate da una qualsiasi applicazione del sistema, realizzando così una maggiore riutilizzazione delle funzionalità implementate; dato il meccanismo visto per i permessi (ogni appiicazione è eseguita all'interno di uno spazio privato), però, questo non può essere fatto in modo diretto, ma viene previsto un meccanismo di comunicazione basato su Intent, particolari messaggi asincroni che l'applicazione invia al sistema, il quale avvierà per suo conto la componente richiesta.

Activities, Services e Broadcast receivers sono tutti attivati appunto tramite Intent; per i primi l'intent contiene l'azione da eseguire e l'URI dei dati su cui va effettuata, mentre per gli ultimi contiene solo un'oggetto che definisce il tipo di messaggio che deve essere trasmesso.

In relazione al real-time, risulta fondamentale approfondire il funzionamento delle Activities e dei Services e di come viene gestito il loro ciclo di vita.

Activities e services

Ogni applicazione può essere composta da molte activities, una delle quali è specificata come principale e mostrata all'utente al primo avvio.

Ciascuna di esse attraversa nel proprio ciclo di vita vari stati intermedi, illustrati in Figura 5.4; i passaggi tra i vari stati vedono la presenza di alcune funzioni di callback, che permettono allo sviuppatore di inserire determinate azioni da intraprendere durante queste transizioni.

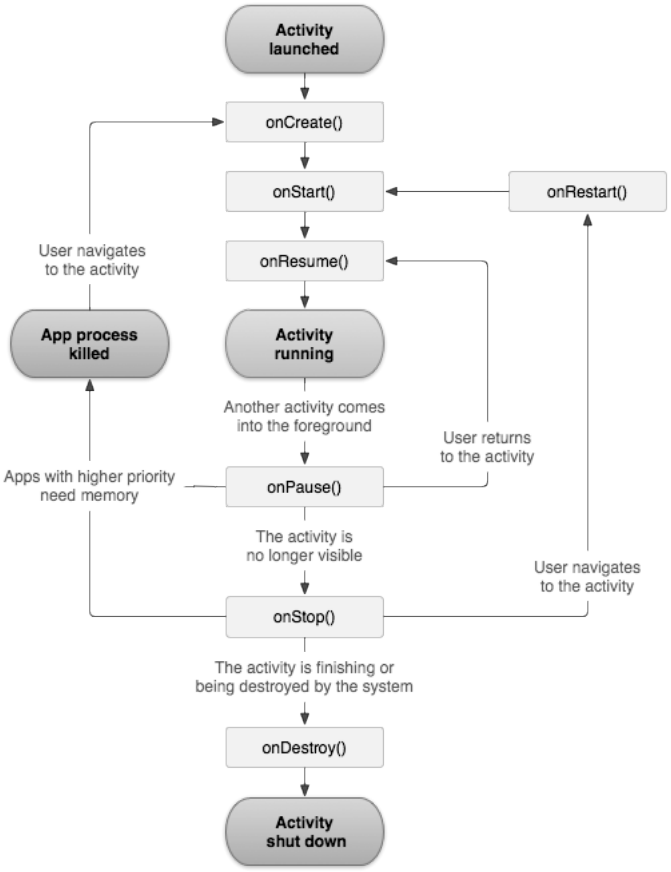


Figura 5.4: Ciclo di vita delle activities

L'intero ciclo può essere visto come l'innesto di tre loop separati:

- **Entire lifetime**: inizia da `onCreate()` e termina alla chiamata della funzione `onDestroy()`. In queste chiamate dovrebbero essere rispettivamente inizializzate le variabili globali dell'applicazione e rilasciate tutte le risorse non più utilizzate. Questo permette di raggruppare in questi frangenti le operazioni generali, che andranno svolte una sola volta nell'intero arco della vita dell'activity.

- **Visible lifetime**: compresa tra `onStart()` e `onStop()`, definisce l'arco di vita in cui l'applicazione è visibile sullo schermo e quindi nelle chiamate vanno definite e rilasciate tutte quelle risorse necessarie a visualizzare l'activity sullo schermo (monitorando ad esempio i cambiamenti che impattano sulla UI).

- **Foreground lifecycle**: in questo ciclo l'activity rimane in primo piano (`onPause()` viene richiamato ad esempio se il dispositivo si pone in stato di sleep o se viene generata una qualche finestra di dialogo); poichè gii eventi di `onResume()` e `onPause()` possono essere molto frequenti, tali funzioni dovrebbero essere mantenute computazionalmente leggere in modo da non provocare un numero elevato di transizioni di stato lente.

Questa gestione particolare del ciclo di vita dell'activity è dunque incentrata nel mantenere in memoria, per quanto possibile, solo le informazioni strettamente necessarie per garantire un più rapido ripristino dell'applicazione nel momento in cui questa torna ad essere in primo piano; se infatti tutte le operazioni di inizializzazione e rilascio delle risorse fossero effettuate in altro modo (ad esempio ogni volta che un'applicazione passa da primo a secondo piano e viceversa), si avrebbero transizioni più lente, un carico elaborativo maggiore e quindi consumi incrementati. L'implementazione di tali stati, se gestita secondo le direttive citate, va a costituire una sorta di cache progressiva dei dati dell'attività, consentendo una maggiore responsività media a fronte di un piccolo incremento nella memoria occupata.

Come si vede, però, lo stesso ciclo di vita prende in considerazione anche la possibilità che, in caso di scarsità di memoria libera le activities in pausa o bloccate possano essere terminate dl sistema per recuperare la memoria necessaria ad applicazioni con priorità maggiore (se l'activity dovrà essere ripresa reinizierà dalla chiamata a `onCreate()`). Per questo le chiamate di `onPause()` e `onStop()` devono prevedere il salvataggio dei dati su supporti persistenti, in quanto l'evento di terminazione sollevato dal sistema può avvenire in modo asincrono (comportando una certa latenza nell'eventuale ripresa della stessa applicazione).

Anche i services hanno un comportamento simile a quello delle activities, ma vengono impiegati per svolgere attività in background senza necessitare quindi di alcuna interfaccia utente. Essi possono essere attivati secondo due modalità differenti:

- `startService()`: il servizio viene lanciato da una componente e sarà eseguito in background per un tempo indefinito, anche nel caso la componente genitrice venga terminata. Il servizio generalmente svolge attività singole e dovrebbe porsi in stato di stop una volta completatele.

- `bindService`: si crea un'interfaccia client-server che consente alle componenti di interagire con il servizio mandando richieste e recuperandone

i risultati forniti. In questo caso il servizio opera fintanto che esiste una componente ad esso collegata, altrimenti viene terminato.

Poichè in ogni caso l'attivazione del service viene gestita come quella di una normale activity, di default viene eseguito all'interno del thread main, e quindi deve essere creato un thread a parte che lo gestisca nel caso esso debba svolgere attività bloccanti o computazionalmente impegnative, per non bloccare il thread principale e incorrere in latenze di risposta indesiderate all'interno dell'interfaccia utente.

5.4.5 Task e back stack

La suddivisione in componenti delle applicazioni comporta che l'utente, nell'esecuzione di un dato programma, richieda al sistema sottostante di far interagire diverse activities, spesso appartenenti ad applicazioni diverse; al fine di rendere invisibile all'utente questa alternanza di componenti garantendogli un'interazione più fluida, in Android sono stati introdotti i *task*.

Un task corrisponde ad un insieme di activities con cui l'utente interagisce per realizzare una data funzionalità; ciascun task è organizzato come uno stack (*back stack*), quindi le varie componenti sono sempre inserite ed eliminate dalla posizione di testa.

La struttura dello stack può evolvere secondo diverse modalità:

- Quando viene eseguita una nuova activity, viene inserita nello stack con un'operazione di push e posta in primo piano, mentre quella precedente viene bloccata (ma ne salva lo stato).

- Ogni qualvolta l'utente preme il tasto Back, l'attività in testa allo stack viene terminata (liberando le risorse ad essa collegate) e quella sottostante torna in primo piano.

- Se invece viene premuto il pulsante Home, che riporta l'utente alla schermata iniziale del dispositivo, l'activity corrente viene bloccata e l'intero task a cui appartiene viene posto in background (mantenendo lo stato di tutte le activities che vi sono contenute). Nel caso venga iniziata una nuova applicazione, sarà creato un nuovo task, mentre se si richiama l'applicazione che aveva dato origine al primo task questo sarà rapidamente riportato in primo piano.

Inoltre, visto che lo stack non subisce mai riordinamenti, se una stessa attività viene richiamata da componenti diverse, questa potrebbe essere istanziata più volte, sia all'interno dello stesso task che in task diversi. Questa politica di funzionamento può essere però modificata, sia a livello di applicazione tramite parametri specifici da inserire nel file `AndroidManifest.xml`, sia attarverso valori passati all'Intent tramite il quale un'activity da il via alla successiva; ad esempio, dichiarando la modalità `singleTask`, se viene

richiamata una componente già presente in uno qualsiasi dei task, sarà quest'ultima ad essere richiamata (in questo modo non si avranno mai activities duplicate).

Il meccanismo adottato consente di ottenere alcuni vantaggi:

- è reso possibile il multitasking pur con la restrizione di avere una sola activity visibile in ogni istante.
- mediamente si ottiene una maggiore responsività dell'applicazione.
- manipolando opportunamente le modalità di gestione dello stack è possibile ottenere migliori latenze o un risparmio di memoria, a seconda delle sequenze di componenti richiamate che si prevedono essere più frequenti.

5.4.6 Processi

Nel momento in cui un componente di una applicazione viene attivato (senza che altri relativi alla medesima applicazione siano già in esecuzione), il sistema Android crea un nuovo processo Linux, composto da un singolo thread.

Di default, tutte le componenti relative ad una applicazione sono eseguite all'interno dello stesso processo, ma un tale comportamento può essere alterato tramite alcuni attributi specificabili nel file `AndroidManifest.xml`, tramite cui si possono specificare quali componenti eseguire in quali processi.

Tra i compiti di Android vi è anche quello di decidere quale processo eliminare: in caso di scarsità di memoria Il sistema da a ciascuno di essi un peso in funzione dell'importanza che il processo può avere per l'utente che lo stà utilizzando; ad esempio, un processo che esegue l'attività correntemente in primo piano avrà maggiore priorità di uno che svolge servizi in background. Il momento in cui terminarlo, quindi, dipende anche dallo stato di tutte le componenti che sono eseguite in quel processo.

Android mantiene ogni processo ad un livello di importanza pari a quello massimo presente tra le sue componenti attualmente attive; se, ad esempio, un processo gestisce sia un service che una activity visibile, sarà quest'ultima a determinarne il livello di priorità. In più, il sistema considera anche la posibile dipendenza tra processi: se il processo A dipende da B, quest'ultimo dovrà avere un'importanza almeno pari a quella del primo.

Tutti questi valori sono codificati rispettivamente nel valore di `importance` e di `importance ReasonCode` presenti nella classe `ActivityManager.RunningAppProcessInfo` e qui sotto riportati:

- `IMPORTANCE_FOREGROUND`: processi in cui viene eseguita un'attività con cui l'utente stà interagendo, un servizio legato a tale attività, un servizio che opera in primo piano, un servizio che stà eseguendo un'operazione di callback (`onCreate()`, `onStart()` e `onDestroy()`) o un broadcast

receiver in cui è stata lanciata la funzione `onReceive()`. Generalmente vi sono pochi di questi processi attivi in un dato istante e sono quelli a priorità maggiore, ovvero gli ultimi a poter essere terminati per recuperare memoria.

- `IMPORTANCE_PERCEPTIBLE` : processi che stanno eseguendo una qualche attività considerata attivamente percepita dall'utente, come ad esempio la riproduzione di un brano musicale in sottofondo.

- `IMPORTANCE_VISIBLE`: sono processi senza componenti in primo piano ma che possono influire su ciò che vede l'utente; sono compresi processi che hanno una activity in stato `onPause()` o un servizio ad esso legata

- `IMPORTANCE_SERVICE`: processi che eseguono servizi lanciati con la chiamata esplicita `startService()`; questi, pur essendo slegati completamente dall'interfaccia, si occupano generalmente di eseguire funzioni quali il download di dati dalla rete e quindi ancora di una certa rilevanza per l'utente.

- `IMPORTANCE_BACKGROUND` : contiene attività nello stato bloccato (dopo la chiamata a `onStop()`) e quindi non hanno alcun impatto sull'interazione con l'utente. Di solito si contano numerosi processi in questo stato, tanto che da venire gestiti tramite una lista LRU (least recently used) in modo da ottenere che quello usato più recentemente sia anche l'ultimo ad essere eventualmente terminato.

- `IMPORTANCE_EMPTY`: sono processi che non possiedono alcuna componente attiva. La ragione dell'esistenza di questa categoria è legata all'utilizzo di tali task come una sorta di c̈acheṗer migliorare il tempo di avvio nel caso l'applcazione venga richiamata ini un secondo momento.

Per quanto riguarda invece le relazioni tra processi, è prevista la seguente classificazione:

- `REASON_PROVIDER_IN_USE`: caso in cui uno dei content provider dell'applicazione sia attualmente utilizzato da un altro processo.

- `REASON_SERVICE_IN_USE`: uno dei services dell'applicazione è correntemente usato da una seconda.

- `REASON_UNKNOWN` : valore di default, non è relativo ad alcuna particolare dipendenza.

I valori precedenti, definiti per ciascun processo, sono poi mappati [50] nel campo `oom_adj` del corrispondente processo nel kernel Linux, dove a valore maggiore corrisponde una probabilità più elevata di essere terminato in caso di scarsità di memoria. La gestione effettiva è affidata al modulo

lowmemorykiller, che sostituisce l'*out-of-memory killer* del kernel Linux in quanto quest'ultimo entra in azione solamente quando la memoria libera disponibile raggiunge un livello critico.

In Android, invece, si ottiene una gestione a grana più fine del comportamento di tale modulo; infatti i sei livelli visti precedentemente corrispondono ad altrettanti valori inclusi nel file
`/sys/module/lowmemorykiller/parameters/minfree` che definiscono i limiti, in pagine di memoria (quindi in unità di 4Kb), oltre i quali il lowmemorykiller inizierà a terminare i processi della categoria corrispondente. Facciamo un esempio per chiarire meglio: se i valori presenti sono 1536,2048,4096,5120,5632,6144, significa che i processi della categoria
`IMPORTANCE_EMPTY` inzieranno ad essere terminati non appena il sistema si trova ad avere meno di 24MB di memoria libera ($6144 * 4/1024 = 24$).

Dalle modalità di gestione delle priorità possiamo evincere che:

- Android mira a mantenere la massima reattività per le componeti a maggiore vocazione interattiva, mirando a ridurre per quanto possibile la latenza percepita dall'utente.

- La priorità dei vari processi è dinamica, poichè dipende sia da quali componenti sono attive, sia dalla presenza di altri processi che dipendono da essi..

- La garanzia di un uso efficace delle priorità è demandata in parte a chi sviluppa l'applicazione. Si può ragionevolmente ipotizzare che una medesima applicazione potrà manifestare prestazioni diverse, a parità di altre condizioni, se ad esempio lo sviluppatore decide di porre tutte le componenti all'interno dello stesso processo o affidandole a processi differenti, in quanto tale scelta influenzerà diversamente le priorità attribuite dal sistema a ciascun processo.

  Inoltre, poichè un processo che esegue un servizio ha maggiore priorità di un altro che esegue attività in background, una activity che inizia una operazione lunga dovrebbe utilizzare un servizio a cui affidarla invece che creare un worker thread (vedi paragrafo successivo), specialmente nel caso in cui l'operazione sopravviva alla stessa attività che l'ha lanciata. Si garantisce così una maggiore priorità all'azione, indipendentemente da come evolve l'esecuzione dell'attività.

- I processi non sono terminati appena si conclude lo svolgimento di una activity, ma solo nel momento in cui il sistema necessita di ulteriore memoria libera: in questo modo successive aperture della medesima activity risulteranno generalmente più rapide, migliorando la responsività media dell'applicazione.

- grazie al lowmemorykiller impiegato si ottiene un adattamento più graduale del sistema all'approssimarsi di una condizione di bassa memoria

disponibile; in questo modo si utilizza in modo meno efficiente la memoria (ne resta mediamente libera una porzione più grande di quella che si avrebbe con il gestore fornito dal kernel Linux) ma dall'altro lato se ne garantisce sempre una minima quantità disponibile, evitando o rendendo comunque meno frequenti i blocchi delle applicazioni per improvvise situazioni di out-of-memory.

5.4.7 Thread

Quando l'applicazione viene lanciata per la prima volta, il sistema crea un nuovo processo con un unico thread, detto *main thread*; questo è un thread critico, in quanto gestisce tutti gli eventi che interagiscono con l'interfaccia utente, incluse le operazioni che ne modificano l'aspetto visivo.

Come comportamento di default, Android esegue tutte le componenti di un processo nel medesimo thread. Nel caso in cui l'applicazione debba gestire componenti che operano elaborazioni computazionalmente impegnative o di durata imprevedibile, come accessi alla rete o interrogazioni di database, queste andrebbero a bloccare per un tempo indefinito il main thread, causando un drastico aumento nelle latenze di risposta dell'interfaccia grafica e quindi un decadimento delle prestazioni. Android prevede due gestori, l'*Activity Manager* e il *Window Manager*, che monitorano la responsività dell'applicazione e, nel caso essa non risponda ad un evento di input entro 5s o un Broadcast receiver non sia completato entro 10s, viene portata in primo piano una finestra di dialogo, indicata con il significativo nome di *Application not responding (ANR)*, tramite la quale si consente all'utente di decidere se arrestare o meno l'applicazione che stà bloccando il sistema.

Per arginare il problema, le componenti che svolgono operazioni temporalmente gravose vanno gestite creando dei thread separati, i *worker thread*; si deve però prestare attenzione al fatto che gli strumenti per gestire l'interfaccia forniti da Android non sono thread-safe, e quindi questi nuovi thread non possono manipolarla direttamente, in quanto potrebbero verificarsi inconsistenze nei dati gestiti (potrebbero generarsi situazioni di accesso concorrente con il main thread).

Android fornisce due approcci possibili per creare nuovi worker thread:

- utilizzare `Activity.runOnUiThread(Runnable)`, `View.post(Runnable)` o `View.postDelayed(Runnable, long)` all'interno di un normale `Thread` Java; in questo modo le comunicazioni tra worker e main thread sono poste in una coda di eventi abbinata a quest'ultimo, che si occuperà di gestirla in modo corretto. Questa soluzione è ottimale per interazioni semplici.

- impiegare la classe `AsyncTask` [43] la quale permette di creare thread asincroni che operano in background e che allo stesso tempo sono in grado di interagire correttamente con il main thread. Ne viene consigliato

Nome	Nice	Modificabile?
THREAD_PRIORITY_DEFAULT	0	SÌ
THREAD_PRIORITY_BACKGROUND	10	SÌ
THREAD_PRIORITY_LOWEST	19	SÌ
THREAD_PRIORITY_URGENT_AUDIO	-19	NO
THREAD_PRIORITY_AUDIO	-16	NO
THREAD_PRIORITY_URGENT_DISPLAY	-8	NO
THREAD_PRIORITY_DISPLAY	-4	NO
THREAD_PRIORITY_FOREGROUND	-2	NO
THREAD_PRIORITY_MORE_FAVORABLE	-1	NO
THREAD_PRIORITY_LESS_FAVORABLE	1	NO

Tabella 5.2: Priorità dei thread in Android

l'utilizzo solo per operazioni di durata massima di qualche secondo, altrimenti conviene gestire l'operazione attraverso un Service.

Nel caso si preveda l'utilizzo di molteplici istanze di `AsyncTask`, si deve tenere presente che queste vengono eseguite in modo sequenziale in un unico thread in background, e quindi si potrebbero creare comunque eccessive latenze. In tal caso, essi vanno creati all'interno di un `ThreadPoolExecutor`, un oggetto al quale possono essere abbinate più operazioni, ciascuna svolta in un thread separato attinto da un insieme di thread creati da un oggetto `Executor`. In questo modo, si ottengono prestazioni mediamente migliori quando si deve gestire un numero elevato di thread asincroni, in quanto viene ridotto l'overhead abbinato all'invocazione di ciascun thread e vengono limitate e gestite le risorse da essi consumate; in più, il `ThreadPoolExecutor` mantiene alcune statistiche, relative ad esempio al numero di thread completati.

Il sistema Android, inoltre, fornisce nella classe *Process* [44], la possibilità di gestire la priorità dei thread, permettendo di controllare il parametro nice dei processi del kernel Linux tramite il metodo `setThreadPriority()`. Sono previsti dieci valori possibili, che distingueremo (vedi 5.2) in un primo gruppo di valori impostabili anche tramite l'applicazione e un secondo gruppo di valori la cui manipolazione può essere fatta solo attraverso questa classe. Per quanto riguarda le priorità video, quella urgent è dedicata ai thread addetti a comporre la schermata di interfaccia utente o a recuperare eventi di input, mentre quella standard è invece attribuita ai thread che ne gestiscono l'aggiornamento. Gli ultimi due valori invece costituiscono gli step minimi con i quali poter incrementare o decrementare la priorità.

Dai dati rilevati possiamo affermare che:

- anche se con alcune restrizioni, lo sviluppatore può intervenire sulle priorità dei thread modificandone quindi il livello di reattività. Questo, se da una parte consente un maggiore controllo sull'applicazione stessa, dall'altra implica che un uso errato di tale strumento possa incidere negativamente sulle prestazioni del sistema. Va ribadito comunque che le funzioni intervengono solo sul livello di nice, mentre ad esempio i processi nativi di Android, diversamente dalle applicazioni in Java, possono essere eseguiti con le priorità real-time del kernel Linux e quindi il funzionamento del sistema di base rimane indipendente da tali problematiche.

- vengono resi disponibili vari strumenti per migliorare la risposta media dell'applicazione, specialmente in presenza di thread multipli eseguiti in secondo piano.

5.4.8 Wakelock

Il consumo energetico è uno dei fattori chiave della fruibilità di un dispositivo embedded; per questo motivo, quando un dispositivo Android è lasciato in stato di riposo, il gestore dei consumi prima diminuisce la luminosità dello schermo, poi lo spegne, fino ad arrestare anche la cpu.

I *wakelocks* [7] costituiscono il meccanismo tramite cui l'applicazione può controllare lo stato del dispositivo ed impedire che questo giunga in uno degli stati sopra elencati; ciò può rivelarsi utile ad esempio nel caso una determinata operazione non debba essere ritardata dal tempo di latenza legato al completo ripristino delle funzionalità del sistema a partire da una condizione di basso consumo.

Poichè un tale intervento può avere un impatto rilevanti sull'aumento dei consumi, l'applicazione deve richiederne esplicitamente il permesso di utilizzo.

Per utilizzarlo, va creato attraverso il metodo `newWakeLock` della classe `PwerManager`, che permette di scegliere fra queste quattro possibili tipologie di wakelock:

- `FULL_WAKE_LOCK`: ripristina il funzionamento di cpu, l'illuminazione massima dello schermo e retroilluminazione della tastiera fisica, se presente.

- `SCREEN_BRIGHT_WAKE_LOCK`: come il precedente eccettuato il controllo della tastiera.

- `SCREEN_DIM_WAKE_LOCK`: mantiene la cpu in esecuzione e lo schermo acceso ma non necessariamente alla massima luminosità. Utilizzato generalmente in applicazioni in cui lo schermo deve rimanere attivo e vi è una bassa interazione con l'utente, ad esempio nella riproduzione di un film.

- `PARTIAL_WAKE_LOCK`: si occupa di attivare la sola cpu. Viene utilizzato quando si vuole impedire che il dispositivo si ponga in stato di sleep prima che una data azione sia stata completata.

Ad essere precisi [45], nell'ultima release rilasciata da Android (la 4.2) i primi tre tipi elencati sono segnalati come deprecati (uno di questi lo è dalla versione 3.2) in favore dell'utilizzo del parametro `FLAG_KEEP_SCREEN_ON` della classe `WindoManager`, ma ancora supportati, considerata anche la presenza di ulteriori due parametri `ACQUIRE_CAUSES_WAKEUP` (forza l'immediata accensione dello schermo) e `ON_AFTER_RELEASE` (mantiene attiva la luminosità per un certo intervallo dopo il rilascio del wakelock), che hanno un effetto reale solo se combinati con i precedenti tipi deprecati.

Per poter utlizzare il wakelock, esso va acquisito dall'applicazione tramite il metodo `acquire()` e successivamente rilasciato con `release()` una volta terminatone l'uso per consentire al sistema di porsi eventulmente in condizioni di basso consumo.

Questa funzionalità di Android rivela quindi la possibilità di incrementare il livello di responsività di una applicazione a fronte di un maggior consumo energetico.

5.4.9 Programma di compatibilità

Considerata la notevole varietà nella composizione e nelle prestazioni dei dispositivi in cui Android può essere installato, Google ha dato vita ad un programma di compatibilità [41] mirato a offrire linee guida uniformi sia per gli sviluppatori che per i produttori degli stessi dispositivi. In modo particolare si desidera:

- fornire le esatte specifiche in termini di funzionalità API e prestazioni che ciascuna versione di Android può fornire se installata in dispositivi che rispettano i requisiti minimi indicati. In questo modo, nonostante la grande varietà di periferiche mobili presenti, gli sviluppatori sono in grado di utilizzare tali informazioni per definire le specifiche delle applicazioni realizzate in modo che esse possano funzionare in modo ottimale su qualsiasi dispositivo compatibile con i vincoli definiti.

- fornire gli strumenti necessari ai sistemi di distribuzione per relizzare filtri appropriati, in modo da presentare all'utente solamente le applicazioni compatibili con le caratteristiche hardware del proprio dispositivo.

- permettere ai vari produttori hardware di differenziare i loro prodotti pur mantenendo al contempo tutti quei caratteri che ne mantengono la comopatibilità con le applicazioni Android.

- minimizzare i costi generalmente associati ai test di compatibilità fornendo alle aziende strumenti gratuiti, come la *Compatibility Test Suite (CTS)* per certificare il rispetto dei requisiti nei prodotti creati.

Per realizzare questi obiettivi, oltre a rendere disponibile il codice sorgente dell'intero sistema in modo da consentire ai produttori un più rapido adattamento dello stesso alle esigenze specifiche più disparate, viene redatto per ciascuna release di Android un documento dettagliato, conosciuto come il *Compatibility Definition Document* [42], nel quale sono enumerati i requisiti che devono essere rispettati dai dispositivi per essere considerati compatibili. Tra le indicazioni fornite, elenchiamo brevemente quelle relative ad aspetti prestazionali del sistema che vanno quindi ad incidere anche sul grado di adattabilità del sistema ad un ambiente soft real-time:

- per ciascuna categoria di dimensione e risoluzione dello schermo di un dispositivo viene definita la memoria minima che la DVM deve riservare a ciascuna applicazione. I quantitativi indicati vanno dai 16 fino ai 128MB per gli schermi di dimensioni superiori ai 960x720 dp e densità di 320dpi.

- per le applicazioni audio, sono consigliate in output(ma non obbligatorie) latenze pari a 100ms o inferiore per il primo frame del flusso audio e una di 45ms o minore per i frame successivi, mentre in input sono indicati rispettivamente 100 e 50ms. Si vuole garantire così una base minimale per le applicazioni mirate a creare effetti audio in tempo reale o gestire comunicazioni VOIP.

- per i vari tipi di sensori sono determinate le frequenze minime con cui questi dovrebbero generare eventi relativi alle grandezze misurate: 120Hz per gli accelerometri, 10Hz per i magnetometri e 200 Hz per i giroscopi, solo per citarne alcuni.

- vengono definiti i requisiti minimi di memoria da riservare per il kernel e lo spazio utente (340MB) e di spazio non volatile disponibile per i dati privati delle applicazioni(350MB) e per lo spazio condiviso (almeno 1GB che deve essere montato in `\sdcard` o raggiungibile con un omonimo link simbolico).

- sono infine riportati come obbligatori i massimi tempi relativi al caricamento iniziale (ovvero il tempo necessario alla creazione del processo nel kernel Linux, al caricamento dei pacchetti Android nella DVM e la chiamata al metodo `onCrate()`) di alcune applicazioni chiave; nel dettaglio, il browser deve essere attivo in meno di 1300ms, mentre per le applicazioni atte a gestire i contatti e la configurazione di sistema sono richiesti al massimo 700ms.

Attraverso le specifiche distribuite si ottengono quindi le seguenti caratteristiche in ambito soft realtime:

- si garantiscono all'utente finale alcuni parametri temporali massimi relativi alle applicazioni che si ritiene siano usate più frequentemente (browser, contatti, ecc) fissandoli come vincoli di progettazione da soddisfare in qualsiasi tipo di dispositivo che desideri essere certificato per eseguire una data versione di Android.
- l'imposizione di quantità minime di memoria per ogni processo fissate in funzione delle caratteristiche dello schermo o deo touchscreen evidenziano una forte connessione tra prestazioni dell'applicazione e gestione dell'interfaccia grafica associata.

5.4.10 Utilizzo di codice nativo

JNI

Nonostante la DVM sia ottimizzata per lavorare su dispositivi con cpu di capacità elaborativa relativamente contenuta, esegue comunque del codice in modo interpretato e dunque con prestazioni velocistiche mediamente inferiori a quelle di codice nativo, compilato per la specifica architettura.

In realtà, Android sfrutta parzialmente l'utilizzo di codice nativo attraverso la *Java Native Interface (JNI)* [46], in quanto lo strato che costituisce il framework delle applicazioni, sviluppato in Java, si appoggia sulle funzionalita fornite dal livello inferiore, composto da librerie in linguaggio nativo.

Il tutto è basato sulla definizione di un insieme di classi di raccordo fra i due contesti, che presentano una interfaccia Java, ma che delegano al codice nativo l'implementazione dei loro metodi; in questo modo, le comunicazioni tra le due parti non sono dirette ma passano comunque attraverso l'intermediazione della macchina virtuale.

Attraverso il *Native development kit (NDK)* offerto da Android viene estesa ad ogni sviluppatore la possibilità di creare e compilare parti delle proprie applicazioni in linguaggio nativo al fine di ottenere prestazioni migliori, generalmente nel caso di funzioni che richiedono un utilizzo intensivo della cpu o per le quali si voglia ottenere un footprint più ridotto. In questo modo è possibile contenere i tempi di esecuzione, a scapito di una minore portabilità dell'applicazione stessa, che potrà funzionare solo sull'architettura per cui la libreria nativa è stata compilata.

JIT

Essendo le appplicazioni programmate in Java, la DVM è dotata di un interprete [51] che scandisce a runtime le varie istruzioni presenti nel file .dex e le trasforma nel codice macchina specifico per la specifica architettura

del dispositivo. Questo passaggio elaborativo è invece assente nei linguaggi compilati, che quindi dimostrano in genere prestazioni superiori a livello di velocità di esecuzione.

Grazie alla buona velocità di elaborazione dell'interprete (circa due volte più veloce di quello della JVM) e alla notevole quantità di codice nativo presente nel sistema operativo e nelle librerie prestazionalmente critiche (che permettono di contenere l'intervento dell'interprete per intervalli pari ad un terzo del tempo di esecuzione totale dell'applicazione), fin dalla nascita di Android non si è ritenuto necessario integrare alcun metodo di ottimizzazione ulteriore.

Per le applicazioni che utilizzano in modo intensivo la cpu, però, il peso dell'interprete può rallentare eccessivamente i tempi di elaborazione (da cinque a dieci volte) e diviene indispensabile la presenza di un un compilatore *Just in time (JIT)*, il quale analizza il codice interpretato, isola le porzioni maggiormente utilizzate e ne mantiene in una memoria cache la relativa traduzione in codice nativo, in modo che le chiamate successive a tali sezioni di codice siano eseguite senza coinvolgere nuovamente l'interprete.

Il JIT può adottare varie strategie di ottimizzazione, decidendo quando effettuarla (all'installazione, all'avvio dell'applicazione, alla chiamata di un metodo, ecc.) o come determinare le porzioni di codice più utilizzate; ogni combinazione di queste mostra vantaggi e svantaggi, e nel caso di Android si è progettata secondo le seguenti direttive:

- Introdurre una minima occupazione di memoria dedicata alla cache del codice nativo.
- Mantenere la coesistenza con il modello di isolamento del codice fornito dalla DVM.
- Ottenere un impatto immediato sulle prestazioni dell'applicazione.
- Evitare interruzioni o rallentamenti nel passaggio da codice interpretato a codice compilato e viceversa.

L'implementazione dei compilatori JIT può seguire due possibili strade:

- **Method JIT**: l'interprete individua i metodi più utilizzati (*hot method*) e ne conserva in cache la versione compilata. Il vantaggio si ha nel fatto che l'esecuzione di codice interpretato e codice compilato è chiaramente distinta poichè le chiamate ai metodi sono sincrone; d'altra parte, ciascun metodo hot può richiamarne al suo interno altri che invece sono utilizzati raramente, e si andrebbero a compilare così anche porzioni di codice che non rispondono ai criteri voluti, con un ulteriore aumento dello spazio di memoria usato per la cache e di risorse elaborative per operare la compilazione.

- **Trace JIT**: individua le sequenze di operazioni più frequentemente utilizzate e le traduce in codice nativo, mantenendole poi in cache. In questo modo il compilatore non è obbligato a rispettare i confini imposti dal metodo (può ottimizzare sequenze che includono solo parte del metodo o che coinvolgono più metodi in sequenza), limitando quindi la sua azione al solo codice hot. Altro vantaggio è la semplicità di realizzazione del compilatore stesso (in caso di situazioni eccezionali delega all'interprete la loro gestione, senza dover integrare quindi le azioni da intraprendere in ogni possibile frangente). Gli svantaggi invece si concretizzano nella necessità d una più frequente sincronizzazione con l'interprete e nel fatto che le sequenze hot cambiano più volte nel tempo ed è quindi più complesso condividerne tra più processi.

In un esempio [51] riportato per evidenziare le differenze quantitative tra i due approcci viene mostrato che, mentre il primo metodo rileva come hot l'8% della dimensione del codice originale, il trace JIT si limita ad individuarne una porzione pari al 2%, con un risparmio della memoria e del tempo cpu che costituiscono l'overhead del compilatore.

Dalle differenze evidenziate è lecito attendersi che Android adotti un compilatore JIT del secondo tipo, come è difatti nella realtà; oltre ai vantaggi precedentemente elencati si ottiene anche un risparmio nei consumi, ottimale per i dispositivi embedded. Nonostante questo, si prevede di implementare anche il supporto al method JIT, che offre prestazioni migliori nei momenti in cui il dispositivo è alimentato da rete e quindi non necessita un risparmio energetico così spinto.

Le fasi del suo funzionamento sono illustrate in Figura 5.5 e qui brevemente riassunte:

1. Vengono definite le istruzioni che possono potenzialmente costituire l'inizio di una sequenza hot (chiamate a metodi, istruzioni if, cicli, ecc).

2. Si incrementa un contatore che tiene traccia del numero di potenziali sequenze.

3. Se la sequenza di istruzioni viene completata entro un determinato limite temporale, si ritorna al punto precedente (la sequenza è troppo breve e compilarla non consentirebbe un guadagno significativo), altrimenti il compilatore verifica se è già stata tradotta in codice nativo.

4. In caso di risposta negativa, attraverso l'interprete viene lanciato un thread per la compilazione e il codice risultante è posto in una memoria cache.

5. Le sequenze tradotte possono eventualmente contenere istruzioni di salto condizionale, di cui si traducono solamente le istruzioni relative al

sotto-caso considerato; si provvede però a marcare l'altro caso in modo tale che, se successive esecuzioni lo richiamano, si torni ad eseguire il punto 3 ed eventualmente si traducano anche queste in una seconda sequenza in codice nativo, che sarà collegata direttamente alla prima.

6. Se invece la traduzione in codice nativo è già stata operata precedentemente, si va semplicemente a richiamarla.

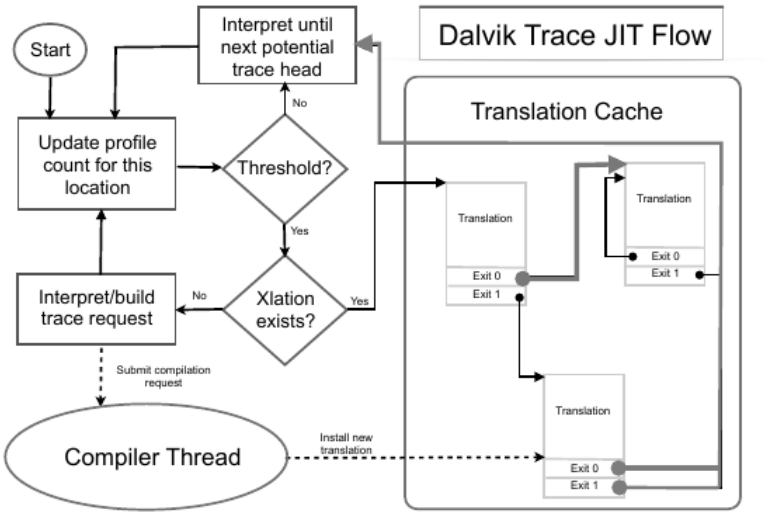


Figura 5.5: Flowchart del compilatore JIT

Nel caso specifico del JIT di Android, vanno evidenziate le seguenti peculiarità:

- Viene mantenuta una cache per il codice nativo ottenuto dal compilatore per ciascun processo, ed essa è condivisibile solo attraverso i meccanismi di sandbox forniti dalla DVM

- L'algoritmo che individua le sequenze di operazioni da ottimizzare è semplice e si limita ad operare prevalentemente sulle porzioni relative a cicli del codice.

Tramite il suo utilizzo, si ottiene un abbattimento dei tempi di esecuzione che va dalle due alle sei volte, con una occupazione di memoria massima di 200KB per processo. Questo però se si parte da un programma totalmente operante in codice interpretato; nei casi reali medi tali guadagni sono in parte ridimensionati per l'utilizzo di funzioni già presenti in codice nativo all'interno del sistema.

Renderscript

L'importanza prestazionale che l'utilizzo di codice nativo riveste all'interno di Android si è concretizzata, a partire dalla versione 4.0, nell'introduzione di *Renderscript* [52], una API che offre un incremento delle prestazioni computazionali, specialmente nell'ambito dell'elaborazione delle immagini o nel calcolo di modelli matematici complessi.

Pur operando a livello nativo, Renderscript deve comunque poter comunicare con la DVM; esso infatti adotta un'architettura di tipo *control and slave*, dove il codice eseguibile a basso livello viene controllato da quello ad alto livello eseguito all'interno della macchina virtuale; più precisamente, la VM mantiene il completo controllo sulla gestione della memoria da allocare a Renderscript, e il framework di Android invoca tramite chiamate asincrone, poste in una coda ed eseguite appena possibile, il codice Renderscript necessario.

La struttura di Renderscript si divide in varie componenti, illustrate in Figura 5.6

- le API Renderscript sviluppate in linguaggio C99 richieste dalla propria applicazione (eseguite a loro volta da un motore di calcolo integratoo in Android).

- Una serie di classi intermedie, generate automaticamente durante il processo di build dell'applicazione dagli strumenti di sviluppo forniti da Android, che fungono da classi wrapper rispetto al livello precedente ed eliminano la necessità di utilizzare codice JNI.

- il framework Android che mette a disposizione le chiamate allo strato intermedio per accedere all'ambiente runtime di Renderscript.

La struttura stessa del Renderscript comporta le seguenti proprietà:

- **Portabilità**: è progettato per operare su dispositivi con tipi differenti di processori (cpu, gpu e dsp) poichè il codice è compilato solo durante il runtime

- **Prestazioni**: fornisce una API per elaborazioni ad alta efficienza che sfruttano tramite parallellizzazione i core presenti nel dispositivo. Tale possibilità viene abilitata attraverso la funzione `rsForEach()`, che automaticamente provvede a determinare i core disponibili nel sistema e a suddividere tra di essi l'esecuzione del codice.

- **Usabilità**: semplifica il processo di sviluppo eliminando la necessità di utilizzare la JNI.

- **Complessità**: incrementa il numero di API da conoscere.

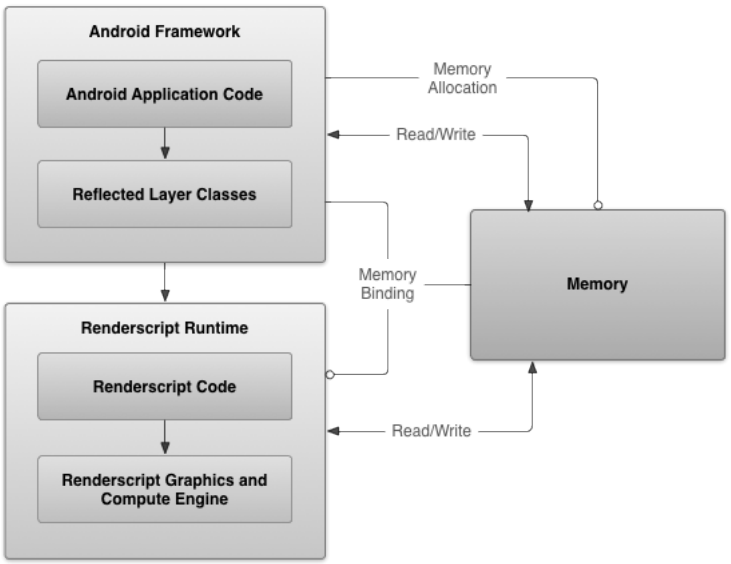


Figura 5.6: Componenti di Renderscript

- **Debug del codice**: potendo operare in parallelo su processori diversi dalla cpu, si complica il debug del codice.

L'attenzione riservata all'ottimizzazione di questa componente mirata a ridurre i tempi di esecuzione è ben evidenziata in alcune analisi prestazionali [53], dove si evidenziano sia i progressi ottenuti nel corso delle versioni 4.0, 4.1 e 4.2 (con velocità più che raddoppiate per elaborazioni delle immagini fatte solo tramite cpu), sia gli incrementi prestazionali (fortemente dipendenti dal tipo di benchmark usato) introdotti dalla 4.2 con la possibilità di utilizzare anche le gpu nel calcolo parallelo.

5.5 Prestazioni real-time

Una prima valutazione [17] di Android è stata condotta su un OMAP 3430 processor, con cuore ARM, generalmente usato in micro-controllori general-purpose.

L'esperimento era mirato alla misurazione della latenza di sistema, ovvero il ritardo combinato introdotto dal kernel Linux e dalla macchina virtuale Dalvik, nella propagazione di un evento di interruzione fino all'applicazione eseguita su Android.

Nella sua realizzazione sono state impiegate due applicazioni distinte:

- **Test**: si limita a creare un'istanza della classe `TimerTask`, la quale permette di lanciare un task dopo un intervallo di tempo predefinito

passato come parametro. Di questo task si osserva l'errore temporale calcolato come la differenza tra l'istante in cui il task viene schedulato dall'applicazione Java e l'istante in cui il timer interrupt viene da questa effettivamente ricevuto.

- **Loading**: anch'essa va a creare un `TimerTask` che periodicamente esegue una funzione contenente varie tipologie di operazioni (calcoli in virgola mobile, chiamate ad API del sistema, ecc) in modo da simulare le comuni attività del sistema; variando la frequenza con cui viene eseguito, si possono quindi ricreare condizioni di carico elaborativo più o meno elevato.

Nello specifico, si sono svolte misurazioni incrociate facendo eseguire il processo Test con periodi di 100ms e 1 ms, mentre si sono simulati un carico nullo, medio ed elevato del sistema portando Loading a non essere eseguito nel primo caso e ad essere eseguito rispettivamente con periodi di 10 e 1 ms negli ultimi due.

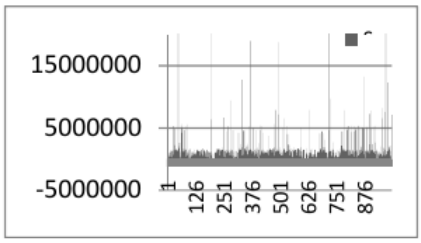
I risultati ottenuti hanno evidenziato alcune problematicità:

- In condizioni di basso carico e applicazione di Test eseguita ad intervalli di 100ms(vedi Figura 5.7), Android tende a portare il sistema in una condizione di basso consumo (*Low power mode*) per risparmiare energia; in questo modo, se una interruzione viene sollevata durante tale periodo, l'intero sistema deve essere prima ripristinato per poterla gestire, comportando un errore che mediamente si aggira intorno ai 10ms.

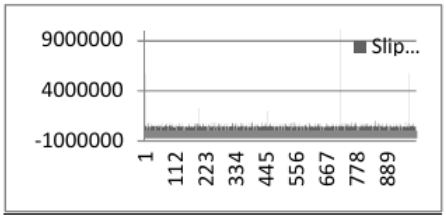
- in condizioni di carico elevato (vedi Figura 5.7), invece, l'errore diviene più contenuto ma si presenta in modo costante; questo perchè, essendo il Loading attivato ogni ms, lo schedulatore deve prima bloccarlo per poter portare in esecuzione il processo Test, generando quindi un ritardo fisso (molto frequente) e costante.

- portando il periodi di Test ad 1 ms, in tutte le condizioni di carico si registra un netto aumento dei ritardi registrati, con una buona percentuale di casi in cui questi toccano i 35, 42 e 80ms rispettivamente per situazioni di carico crescente.

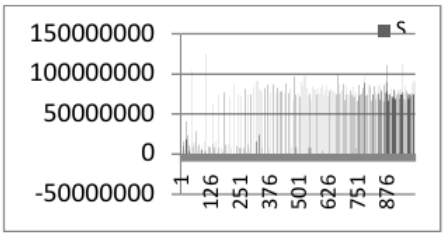
- sempre nelle condizioni di Test con periodo di 1ms (vedi Figura 5.7), si osserva un fenomeno di accumulo progressivo del ritardo; con il passare del tempo considerato dall'istante di avvio dell'applicazione, la frequenza dei ritardi tende ad aumentare, e questo può comportare in situazioni reali un decadimento delle prestazioni e della responsività di sistema in presenza di applicazioni che devono gestire interruzioni a frequenze superiori al KHz e operare per un certo lasso di tempo.



(a) *Basso carico e Test a 100ms.*



(b) *Alto carico e Test a 100ms.*



(c) *Alto carico e Test a 1ms.*

Figura 5.7: Misurazioni latenze Android in varie condizioni di carico

Capitolo 6

Ubuntu phone

6.1 Sommario

In questo capitolo sono raccolte le caratteristiche del nuovo sistema operativo per dispositivi mobili Ubuntu Phone, operando simultaneamente un confronto con quelle di Android mirato a determinare qualitativamente quali scelte progettuali realizzate nei due sistemi siano in grado di offrire migliori prestazioni e una più efficace responsitività all'interno dei dispositivi portatili da questi gestiti.

6.2 Storia

Ubuntu Phone OS, il sistema operativo di Canonical per smartphone è stato presentato [60] dall'8 all'11 gennaio al Consumer Electronics Show di Las Vegas e ha attirato l'attenzione di molti media e fan del settore. All'evento le principali funzionalità del sistema sono state mostrate attraverso l'installazione di una sua preview su uno smartphone Galaxy Nexus; dalla presentazione se ne evince principalmente il diverso approccio alla gestione del'interfaccia rispetto agli altri concorrenti sul mercato, di cui andremo successivamente a dare maggiori dettagli, e la possibilità per i dispositivi più potenti di essere trasformati, tramite una dock station, in un vero e proprio sistema desktop completo.

A partire dal 21 febbraio sono state poi rilasciati i binari del sistema solamente per alcuni dispositivi, in particolare Galaxy Nexus, Nexus 4, 7 e 10; i primi smartphone sul mercato [61] però non arriveranno che nel 2014, in quanto sono necessari un ulteriore sviluppo del codice e una approfondita fase di test sui vari dispositivi prima dello sbarco definitivo sul mercato.

Canonical non ha ancora dichiarato ufficialmente se il progetto sarà reso opensource; per ora sono state rilasciate solo immagini binarie, mentre il codice sorgente è disponibile solo per gli sviluppatori.

6.3 Struttura

Uno schema di massima della struttura del sistema operativo Ubuntu, che ne raccoglie le componenti principali senza però evidenziare con quali modalità esse interagiscono, è presentato nella Figura 6.1, dalla quale si possono distinguere i seguenti blocchi:

- Alla base è sempre presente il kernel Linux.
- Librerie native di base per grafica, sicurezza, multimedia, configurazione, storage dei dati, accessibilità, comunicazioni di rete, internazionalizzazione e ubuntu one.
- Componenti dell'interfaccia utente.
- Un'ampia varietà di strumenti di base per sviluppare la piattaforma, come librerie Qt, GTK+, XUL e HTML5.
- Il supporto a molteplici linguaggi di programmazione, come Python, C, C++, Javascript, Java, QML, Vala e C#.
- L'insieme delle applicazioni utente in corredo con il sistema.

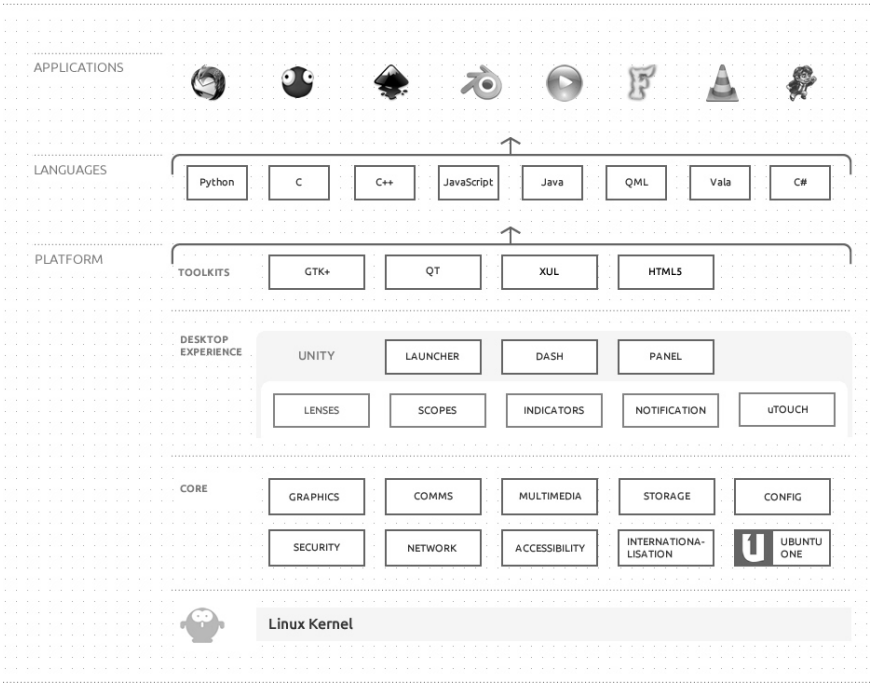


Figura 6.1: Struttura del sistema operativo Ubuntu

Componenti	Requisiti minimi	High-end smartphone
Cpu	**1Ghz Cortex A9**	**Quad-core A9 or Intel Atom**
Memoria	**512MB – 1GB**	**Min 1GB**
Spazio di storage	**4-8GB eMMC + SD**	**Min 32GB eMMC + SD**
Multi-touch	**SI**	**SÌ**
Estensione desktop	**NO**	**SÌ**

Tabella 6.1: Requisiti hardware per Ubuntu phone

6.3.1 Dispositivi target

Mentre Android è ottimizzato esclusivamente per smartphone e tablet, la nuova versione del sistema operativo Ubuntu mira ad ottenere un ambiente unico capace di operare e adattarsi a qualsiasi tipo di sistema, dai dispositivi di telefonia a basso livello fino ai sistemi desktop.

Gli stessi requisiti hardware richiesti per l'installazione di Ubuntu phone (vedi 6.1) dimostrano, almeno nelle intenzioni, due obiettivi ben specifici:

- Garantire un soddisfacente livello di funzionalità e interazione anche nei dispositivi di basso livello; i requisiti minimi indicati sono attualmente soddisfatti dalla quasi totalità dei dispositivi.
- Per i dispositivi con capacità elaborativa più elevata (indicati come *superphone*), si vuole garantire la possibilità di utilizzarli, tramite una dock sstation, un monitor e una tastiera, anche come dei veri e propri pc desktop completi.

Ubuntu phone OS si avvantaggia anche dei profondi miglioramenti hardware avvenuti negli ultimi anni in campo mobile, potendo supportare cpu con core multipli e sfruttare le potenzialità delle gpu attraverso le librerie OpenGL e GLES, mentre Android era stato inizialmente pensato per adattarsi ad ambienti con risorse di calcolo ben più limitate.

Sotto l'aspetto dei vincoli temporali legati ai tempi di risposta dei dispositivi, Android adotta però un approccio più rigoroso, fornendo per ciascuna sua versione il documento CDD con il quale si offrono al consumatore maggiori garanzie sulla responsività e sulle reali prestazioni del dispositivo che si andrà ad acquistare.

6.3.2 Kernel

Il kernel utilizzato sarà sempre quello Linux adattato [65] per funzionare su chipset con ARM e Intel x86 con il sistema centrale basato su *Support Package Android Board (BSP)*; questo significa che i produttori hardware e OEM che già hanno in commercio dispositivi Android non dovranno fare nessuna operazione complicata di porting, in quanto i driver per le varie periferiche sono già inclusi nel BSP citato. Sotto questo profilo, quindi, non

dovrebbero comparire sostanziali differenze nel comportamento temporale rispetto a quello riscontrabile in Android.

6.3.3 Esecuzione dei programmi

A differenza del sistema operativo di Google, in Ubuntu phone non viene utilizzata alcuna macchina virtuale, eliminando quindi le latenze ad essa legate ed affidandosi alla maggiore velocità di esecuzione del codice nativo; con questa scelta, di conseguenza, vengono anche a cadere le problematiche relative alla variabilità nei tempi di latenza introdotta dagli interventi imprevedibili del garbage collector.

Malgrado quindi entrambi i sistemi siano basati sullo stesso kernel, le applicazioni scritte per Android non saranno in alcun modo compatibili col il nuovo sistema, mancando la Dalvik VM in grado di interpretare ed eseguire il codice.

La maggior parte delle applicazioni incluse [58] quindi, derivano direttamente da quelle realizzate per Linux, nelle quali è stata però ridisegnata l'interfaccia grafica per adattarla ad operare su dispositivi mobili; per realizzare ciò, i progettisti di Ubuntu phone hanno fatto ricadere la loro scelta sull'adozione delle librerie Qt e del linguaggio QML, che andremo ad approfondire nei paragrafi immediatamente successivi.

Per ottenere una maggiore compatibilità, Ubuntu phone prevede anche il supporto a tutte le *web application*, programmi che vengono eseguiti direttamente all'interno di un browser e sono quindi indipendenti dalla piattaforma sottostante; questi ultimi, pur garantendo quindi la massima portabilità, ne pagano uno scotto in termini di prestazioni, risultando meno efficaci rispetto alle applicazioni native. Le novità introdotte dall'HTML5 [62] rispetto all'HTML 4 sono finalizzate soprattutto a migliorare il disaccoppiamento tra struttura, definita dal markup, caratteristiche di resa (tipo di carattere, colori, eccetera), definite dalle direttive di stile, e contenuti di una pagina web, definiti dal testo vero e proprio. Inoltre l'HTML5 prevede il supporto per la memorizzazione locale di grosse quantità di dati scaricati dal web browser, per consentire l'utilizzo di applicazioni basate su web (come per esempio le caselle di posta di Google o altri servizi analoghi) anche in assenza di collegamento a Internet.

6.3.4 Librerie Qt

Qt è contemporaneamente una libreria di classi C++ ed un insieme di strumenti per lo sviluppo di interfacce utente grafiche (GUI) multi piattaforma, e' conosciuto infatti anche con il nome di toolkit Qt. Sviluppato dalla norvegese Trolltech, Qt permette lo sviluppo software secondo la filosofia "write once, compile anywhere", che significa letteralmente "scrivi una sola volta e compila ovunque". In altre parole, Qt permette agli sviluppatori di scrivere

codice sorgente perfettamente compilabile su diversi sistemi operativi, quali ad esempio MS Windows, Mac OS X, Linux, Solaris, HP-UX, molte versioni UNIX con grafica X11 ed embedded Linux. Cio' e' stato reso possibile grazie all'adozione di un'interfaccia di programmazione delle applicazioni (API) unica ed indipendente dall'hardware e dal software di sistema. Insieme con la libreria di classi, vengono forniti strumenti di supporto per il design grafico (Qt Designer), per la traduzione linguistica (Qt Linguist) e il manuale in linea (Qt Assistant).

Per sistemi UNIX, Qt rappresenta un'ottima scelta per lo sviluppo:, è portabile, è veloce, facile da imparare ed usare, inoltre, se si scrive software open source per Linux, BSD, Solaris o molte altre varianti UNIX, Qt è gratuito; in altre parole, non si deve pagare alcuna licenza.

Un toolkit che permetta lo sviluppo di GUI multi-piattaforma, deve implementare una strategia per nascondere al programmatore i dettagli dovuti alle API native del sistema operativo sottostante.

Esistono fondamentalmente tre strategie differenti adatte allo scopo:

- **API layering**: prevede la costruzione di una nuova API fornita da uno strato software che viene localizzato al di sopra della API nativa. I vantaggi offerti da questa soluzione sono una relativa semplicità di programmazione ed una totale conformità con il look&feel (aspetto estetico) nativo. Per contro lo svantaggio principale è dovuto alla lentezza dei programmi scritti, basandosi sulle nuove API; infatti lo strato software aggiuntivo va ad appesantire l'esecuzione con inevitabili rallentamenti.

- **API emulation**: consiste nell'emulare le API di un unico sistema su tutte le altre piattaforme. In questo caso, non occorre innestare uno strato software al di sopra delle API native per la piattaforma emulata, occorre bensì inserirla per le altre piattaforme, diverse da quella usata come riferimento. Benché questa soluzione sembri risolvere i problemi di portabilità, in realtà è difficile da ottenere, in quanto se le piattaforme sono tra loro troppo differenti, l'uso di una API di riferimento è molto complicato da realizzare.

- **GUI emulation**: adottata da Qt e consiste nell'emulare il vero comportamento di una GUI; per fare ciò, vengono usate solamente le primitive grafiche di base, offerte da ciascuna piattaforma, quali ad esempio le funzioni per tracciare un punto, una linea, un cerchio. Il vantaggio di questa soluzione è evidente, non si appesantisce l'applicazione con uno strato software aggiuntivo e neppure si devono uniformare tutte le API dei sistemi su cui garantire la portabilità. Lo svantaggio principale di questa soluzione è dovuto al fatto che l'emulazione della GUI deve essere fatta in modo da eguagliare il più possibile il look&feel della piattaforma nativa ed inoltre, quando anche solo un nuovo widget

(componente di GUI) viene aggiunto o modificato su una piattaforma, il toolkit deve essere aggiornato e riemesso (la possibilità di modifiche o aggiunte ai widget grafici di una piattaforma consolidata è comunque un evento assai raro). La GUI emulation permette quindi al toolkit Qt di emulare il comportamento di ogni singola GUI per ogni sistema operativo supportato e di fornire quindi al programmatore un ambiente unico, svincolato dalla piattaforma su cui verrà eseguito, basandosi su un'unica API che nasconde i dettagli sottostanti. Per portare un'applicazione da una piattaforma ad un'altra, sarà sufficiente ricompilare il codice sorgente sulla nuova piattaforma e mandare in esecuzione il codice eseguibile generato.

Il punto cruciale a cui si doveva trovare soluzione, era quello di far comunicare tra loro gli oggetti C++ della loro libreria, senza per questo dover dipendere dai metodi di comunicazione nativi (quali ad esempio la gestione degli eventi di MS Windows o di X11), in modo da rimanere indipendenti dall'implementazione.

Supponiamo di avere una finestra di dialogo contenente due oggetti grafici molto semplici, un bottone ed un indicatore visivo (immaginiamo, a titolo di esempio, qualcosa di simile alla lampada di un semaforo, di colore rosso) e supponiamo inoltre di voler dotare la nostra finestra di una semplice funzionalità: quando viene eseguito un click del mouse sul bottone grafico (in altre parole, il bottone viene "premuto"), il colore del nostro semaforo diventa verde. Il semplice esempio sopra riportato, ci pone di fronte al problema di far comunicare tra loro i nostri due oggetti grafici: il pulsante ed il semaforo; in altre parole, quando il pulsante viene premuto, deve essere emesso un "segnale" al semaforo, in modo che esso possa riconoscere l'evento e cambiare colore.

In estrema sintesi, il nostro problema di comunicazione può essere descritto con le parole di "segnale", corrispondente all'evento di bottone premuto e "cambio colore", funzione propria dell'oggetto grafico semaforo, innescata dal segnale di bottone premuto (usando la terminologia Qt, questo è uno "slot"). Allo stato attuale delle cose, abbiamo identificato un segnale ed uno slot, ciò che ancora ci manca è un metodo per il loro collegamento.

Gli slots sono praticamente identici alle funzioni membro di una classe, si può quindi parlare di slots pubblici, privati e protetti secondo l'accezione classica del linguaggio C++ e possono essere invocati così come avviene per tutti gli altri metodi tradizionali di una classe. La differenza sostanziale è che uno slot, può sempre essere collegato ad un segnale e quindi sarà invocato ogni volta che il segnale verrà emesso. Questo collegamento signal/slot, viene realizzato con la funzione di libreria connect ed ha la sintassi: `connect (sender,SIGNAL(signal),receiver,SLOT(slot));` dove sender e receiver sono i puntatori agli oggetti Qt (nel nostro esempio sender è il bottone e receiver il semaforo) da porre in comunicazione e signal, slot

sono funzioni (in questo caso senza parametri) appartenenti rispettivamente all'oggetto sender e receiver.

La macro `SIGNAL()` e la macro `SLOT()` sono essenziali per il funzionamento del tutto e verranno pre-processate dal meta object compiler (moc) prima della compilazione di tutti i files del progetto. Il meta object system di Qt è quindi un passo di pre-processing in grado di generare codice C++ puro, partendo dalla definizione degli oggetti Qt. In questo modo, il meccanismo signal&slot, potrà funzionare su qualsiasi sistema operativo, l'unica condizione necessaria è quindi la disponibilità di un compilatore C++ standard (quale ad esempio gcc). L'uso di moc e qmake (strumenti per la compilazione dei programmi realizzati con Qt) libera quindi il programmatore da tutti i dettagli di implementazione necessari al funzionamento "multipiattaforma" del meccanismo signals e slots. Così come esiste una funzione connect, esiste anche la funzione opposta, la `disconnect`, usata per disconnettere il segnale dallo slot precedentemente collegati.

In questo modo è possibile attivare e disattivare il collegamento tra oggetti a run-time. Va inoltre ricordato che più segnali possono essere collegati ad uno stesso slot, un segnale può essere collegato a più slots ed infine un segnale può essere collegato ad un altro segnale:
`connect (sender,SIGNAL(signal1),receiver,SLOT(signal2))`; in questo caso, quando viene emesso il segnale `signal1` , verrà anche emesso il segnale `signal2` .

In questo modo, oltre ad ottenere una maggiore portabilità, il meccanismo garantisce inoltre una gestione tipizzata delle comunicazioni (la firma del metodo che corrisponde al segnale deve corrispondere a quella dello slot che lo riceve). Si ottiene così il vantaggio di ottenere interfacce grafiche di estrema portabilità mantenendo le prestazioni offerte dall'utilizzo di codice nativo.

6.3.5 Interfaccia utente

I principi cardine [59] che dovrebbero stare dietro alla realizzazione delle applicazioni per Ubuntu sono tre:

- Dare ai contenuti il massimo spazio e la maggiore visibilità, mentre tutto il resto deve rimanere nascosto oltre i bordi dello schermo.

- Le interazioni sono basate su gesture ai limiti dello schermo, e devono offrire una interazione naturale che richieda all'utente il minimo sforzo per recuperare i dati o le applicazioni desiderate. Le interazioni devono essere fluide e consentire all'utente dove si stà muovendo e cosa stà succedendo.

- ogni layout, forma, carattere tipografico o ombra è scelta attentamente e deliberatamente impostata; le caratteristiche dei font di ubuntu

sono state applicate all'interfaccia per ottenere un linguaggio visuale uniforme.

I primi due punti sono realizzati usando swiping gestures sui margini dello schermo, in modo tale da rendere tutti i menù a scomparsa e garantire ai contenuti l'occupazione dell'intero schermo; ognuno dei quattro margini assolve ad una determinata funzione:

- Il margine superiore contiene i sistemi di servizio (orologio, volume, connessione di rete, livello di carica) e i menù per le impostazioni.
- Il margine destro consente. tramite ciascuna interazione, di tornare indietro di un passo nella sequenza delle applicazioni utilizzate. Viene così eliminata la necessità di avere un tasto Home, necessario su Android per poter eseguire esplicitamente una nuova applicazione.
- Il margine inferiore fa comparire i controlli dell'applicazione corrente. Inoltre, grazie alla presenza di un pulsante back, consente di scorrere all'indietro le viste dell'applicazione corrente attraversate precedentemente.
- Il margine sinistro invece nasconde un menù contenente collegamenti alle applicazioni preferite, visualizzabili con uno swipe corto: uno swipe completo da sinistra verso destra permette invece di visualizzare tutte le applicazioni correntemente aperte.

Ogni applicazione deve fornire almeno due layout: un *core layout* che comprende una barrà dei menù in alto, un separatore, una content area principale e una toolbar inferiore nascosta che si rivela interagendo con il bordo inferiore, e un *full screen layout* dove il menù superiore si rivela solo interagendo con il margine superiore dello schermo (lasciando quindi l'intero spazio disponibile per i contenuti). Deve essere possibile lanciare direttamente l'applicazione nel layout di tipo full screen o in uno di tipo *full screen view*, ovvero quando una singola porzione di contenuto nella schermata principale dell'applicazione viene portata all'attenzione dell'utente estendendola ad una visualizzazione a tutto schermo.

La navigazione attraverso le applicazioni o le viste interne a ciascuna di esse viene realizzata attraverso alcune strutture di base:

- **Page stack**: viene utilizzato quando ci si deve spostare dalla pagina principale ad una secondaria dell'applicazione. In questo caso, per ritornare alla precedente va richiamato il menù in basso e si utilizza il pulsante *back*. Questo può essere ripetuto per ogni sottolivello di pagina raggiunto, e il back è utilizzabile fino al raggiungimento della schermata iniziale. La sua realizzazione avviene attraverso la struttura `PageStack`, che prevede per ora solo i seguenti tre metodi:

  - `push(page, properties)`: inserisce una nuova pagina nello stack, applicandone opzionalmente le proprietà fornite.
  - `pop()`: estrae un elemento dallo stack. La documentazione, ancora in via di realizzazione, risulta contraddittoria; si precisa infatti che l'estrazione avviene se la dimensione dello stack è almeno pari ad 1, ma subito dopo si dichiara che la funzione non fa nulla nel caso siano presenti o 0 o un solo elemento.
  - `clear()`: disattiva la pagina correntemente visualizzata e svuota lo stack.

- **Flat**: usata per spostare l'utente tra viste principali di uguale importanza, usando la struttura delle *tab*, poste sempre nell'header dell'applicazione

- **Contextual**: l'utente si muove tra differenti livelli di dettaglio con una sola vista, ed è realizzata tramite la *expansion*, che permette di espandere il contenuto dell'oggetto selezionato (per tornare indietro o toccare all'esterno del contenuto espanso o fare il pinch). In questo caso la navigazione è limitata a due soli livelli (indice dei contenuti e visualizzazione espansa), se ne servono di ulteriori va utilizzato il page stack.

- **Deep**: l'utente si muove attraverso i livelli gerarchici in cui è suddivisa l'applicazione, con l'ausilio dello page stack visto prima.

Non sono stati rilasciati dettagli in merito alla possibilità di gestire il ciclo di vita delle applicazioni; si può ipotizzare però che, essendo l'integrazione di tutte le tipologie di dispositivi uno dei punti cardine delle future versioni di Ubuntu, esso si discosterà da quello sviluppato su Android, presentando probabilmente le seguenti differenze:

- Mancanza di una suddivisione dei processi nelle categorie di priorità definite da Android; probabilmente verrà mantenuta la gestione di priorità di base offerta dal kernel Linux.

- Non sarè presente nemmeno la suddivisione tra applicazioni visibili o in background, eliminando le problematiche di salvataggio dello stato e di mantenimento degli stack ma andando quasi sicuramente ad avere una maggiore occupazione di memoria.

- È possibile passare da un'applicazione all'altra senza necessariamente passare per la pagina home, grazie alla presenza del menù a scomparsa posto sul lato sinistro; la pagina home risulta ancora presente ma dedicata alla visualizzazione delle ultime applicazioni usate, di eventuali notifiche e di un condensato dei contenuti multimediali memorizzati nel dispositivo.

- La chiusura delle applicazioni non viene gestita autonomamente dal sistema, ma lasciata al controllo diretto dell'utente tramite il tasto di terminazione presente nel menù dell'applicazione stessa. La scelta è giustificata dal fatto che il nuovo sistema operativo, oltre a voler mantenere la stessa funzionalità di interazione con i programmi presente nei sistemi desktop, viene ad essere sviluppato in un momento in cui i requisiti hardware, in particolare la quantità di memoria disponibile, non hanno più i limiti stringenti presenti invece al momento del rilascio dl sistema Android.

6.4 Prime valutazioni

Non essendo in possesso di un dispositivo tra quelli per i quali è stata rilasciata un'immagine del nuovo sistema operativo, per le prime valutazioni ci possiamo basare solo su alcuni test di carattere qualitativo [63], che dimostrano come, nonostante la versione rilasciata sia solo una preview, il sistema risulta sufficientemente reattivo almeno a livello di utilizzo dell'interfaccia utente; inoltre, sembra risultare più veloce della corrispondente controparte dedicata ai tablet, *Ubuntu Tablet*, per la quale la stessa Canonical indica però requisiti minimi più elevati.

Ulteriori dati saranno sicuramente disponibili a breve grazie anche ad immagini del sistema aggiornate quotidianamente e all'estensione del range di dispositivi per i quali saranno rilasciate [64], tra cui Asus Transformer, HTC Desire, Kindle Fire, Motorola RAZR, Sony Xperia e molti altri.

Capitolo 7

Conclusioni

L'ampia diffusione di dispositivi elettronici di largo consumo impiegati prevalentemente per la fruizione di contenuti multimediali, ha visto negli ultimi anni un accresciuto interesse per la gestione di applicazioni operanti in tempo reale nelle quali i vincoli temporali devono essere rispettati secondo criteri statistici e non in termini assoluti. Questo nuovo campo ha quindi necessitato di nuovi criteri per definire la qualità del servizio offerto dalle applicazioni stesse, lo studio delle problematiche ad esso legate e l'ideazione di algoritmi atti ad eliminarle o quantomeno a ridurne gli effetti in base a predefiniti parametri di valutazione.

Il kernel Linux, benchè nato prevalentemente per essere eseguito su calcolatori general-purpose, ha acquisito nel tempo tutta una serie di caratteristiche capaci di migliorarne le prestazioni anche all'interno dei dispositivi embedded soft real-time, dove diviene cruciale un buon bilanciamento tra occupazione delle scarse risorse disponibili e latenze di risposta agli input ricevuti dall'ambiente; tra le principali, ricordiamo il supporto alla prelazione anche per i processi del kernel, la gestione unificata di processi e thread, le numerose primitive di sincronizzazione, l'ottimizzazione del processo di fork mirata a ridurre l'occupazione di memoria e il tempo di attivazione del processo stesso, le varie strutture disponibili per la gestione delle interruzioni e i meccanismi di allocazione e di caching della memoria mirati a ridurre l'overhead per la sua manipolazione e per il recupero dei dati immagazzinati.

Tuttavia, le scelte progettuali implementate sono state indirizzate in modo particolare alla riduzione dei tempi di esecuzione o al miglioramenti dei tempi medi di risposta, senza di fatto occuparsi dell'aspetto della gestione di eventuali vincoli temporali legati ai processi; di questo fatto sono testimonianze la mancanza di parametri per registrare le eventuali deadline (anche di tipo soft) associate ai processi o il mantenimento di algoritmi di schedulazione che favoriscono un'equa distribuzione delle risorse elaborative della cpu piuttosto che una maggiore responsività delle applicazioni.

La sua diffusione di utilizzo ha però di fatto sancito che le peculiarità

del kernel Linux possono essere efficacemente sfruttate in ambito soft real-time anche attraverso approcci completamente diversi quali quelli adottati dai sistemi operativi presi in esame, Android e Ubuntu phone.

Il primo, pur adottando un linguaggio interpretato come Java per lo sviluppo delle applicazioni e quindi aggiungendo con l'intervento della macchina virtuale e del garbage collector elementi ulteriori capaci di influenzare in modo negativo le latenze di risposta del sistema, ha saputo contenere queste limitazioni attraverso l'ottimizzazione spinta della DVM e dell'occupazione della memoria, l'utilizzo di librerie di base native, l'adozione di nuove modalità di gestione del ciclo di vita delle applicazioni e la gestione delle priorità fortemente legata alla componente grafica di interazione con l'utente.

Il secondo sistema, anche se ancora distante dall'assumere la sua forma definitiva, presenta marcate differenze rispetto al precedente, legate soprattutto all'eliminazione della macchina virtuale in funzione di un approccio più fortemente legato al codice nativo, all'utilizzo delle librerie Qt per la realizzazione delle interfacce grafiche, all'adozione di una struttura diversa nella impostazione dei menù di interazione con l'utente e alla volontà di realizzare un ambiente in grado di fornire un'esperienza del tutto simile a quella dei sistemi desktop, almeno nei dispositivi più potenti. Si evidenzia inoltre una minore attenzione al risparmio della memoria, giustificata in parte anche dall'attenuazione dei vincoli sulle risorse disponibili nei dispositivi embedded verificatasi in questi ultimi anni.

7.1 Sviluppi futuri

Vista la fase embrionale di Ubuntu phone OS successivi lavori potranno concentrarsi su una più dettagliata descrizione dello stesso, andando a colmare quegli aspetti qui riportati solo come ipotesi o fondati su dati non ancora definitivi.

Inoltre, si renderebbero necessari ulteriori test riguardanti le latenze del sistema operativo Android, in quanto i pochi recuperati sono relativi a versioni oramai vecchie e sono stati effettuati su dispositivi particolari e non direttamente su prodotti commercializzati; dato il notevole grado di evoluzione della piattaforma, le considerazioni sulle latenze qui riportate potrebbero non essere più valide per le versioni attuali. In questo modo, affiancandoli a misurazioni documentate delle prestazioni di responsività della piattaforma Ubuntu phone, si sarà in grado di definire un quadro di confronto più oggettivo e realistico.

In aggiunta, potrebbe rivelarsi interessante sostituire alcune componenti del kernel Linux di base (in modo particolare l'algoritmo di schedulazione di base) o ricompilare il kernel con differenti opzioni di schedulazione per andare a valutarne quantitativamente gli effetti sulla responsivita dei sistemi operativi descritti in questa sede.

Elenco delle tabelle

5.1 Dimensioni occupate dai vari formati di bytecode 106
5.2 Priorità dei thread in Android 119

6.1 Requisiti hardware per Ubuntu phone 133

Elenco delle figure

2.1 Parametri temporali dei job 5
2.2 Funzioni di utilità associate ai vari tipi di job 10

3.1 Tempi di decodifica per una sequenza di frame video (i tempi di decodifica sono espressi in microsecondi) 20
3.2 EDF fallisce nel garantire la protezione temporale 33
3.3 CBS garantisce la protezione temporale anche con arrivi ritardati. 33
3.4 Calcolo del budget dei residui 37
3.5 Situazione di forte posticipo della deadline per il primo server 46
3.6 Impatto del server CBS con advancing deadline 46

4.1 Struttura generale del kernel 57
4.2 Relazioni fra gli stati dei task 64
4.3 Andamento delle occorrenze di RCU nel kernel Linux 88
4.4 Strati software presenti tra una funzione in spazio utente e il dispositivo fisico di archiviazione dati 95

5.1 Il framework software di Android 103
5.2 Conversione dal bytecode di Java a quello adottato dalla DVM 105
5.3 Il processo Zygote 108
5.4 Ciclo di vita delle activities 112
5.5 Flowchart del compilatore JIT 126
5.6 Componenti di Renderscript 128
5.7 Misurazioni latenze Android in varie condizioni di carico . . . 130

6.1 Struttura del sistema operativo Ubuntu 132

Bibliografia

[1] BUTTAZZO, G., *Hard real-time computing systems, 3ed*, 2011 cap. 2

[2] BUTTAZZO, G., *Soft real-time systems*, 2005

[3] LIU, J. W. S., *Real-time systems*, 2000, pag. 26-34

[4] LOVE, R., *Linux kernel development*, 2010

[5] RAGHAVAN, P., LAD, A., NEELAKANDAN, S., *Embedded Linux system design e development*, 2006, pag 202-209

[6] BOVET, D. P., CESATI, M., *Understanding the Linux kernel, 3rd edition* 2005, cap. 6.1

[7] MEIER, R.,*Professional Android 4 Application development* 2012, pag 757-758

[8] MCKENNEY, P., *SMP and embedded realtime myths*, http://www2.rdrop.com/ paulmck/realtime/SMPembedded.2006.10.09b.pdf 09/10/2006 (visto il 15/01/2013)

[9] LIPARI, G., ABENI, L., *A bandwidth ihneritance algorithm for real-time task syncronization in open systems*, Dicembre 2001

[10] CHENYANG LU, JOHN A. STANKOVIC, GANG TAO, SANG H. SON, *Design and evaluation of a feedback control EDF scheduling algorithm*, 1999

[11] LIPARI, G., BUTTAZZO, G., *Scheduling real-time multi-task applications in an open system*, Giugno 1999

[12] FAGGIOLI, D., CHECCONI, F., TRIMARCHI, M., SCORDINO, C., *An EDF scheduling class for the Linux kernel*, 2009

[13] FAGGIOLI, D., CHECCONI, F., TRIMARCHI, M., *An implementation of the earliest deadline first algorithm in Linux*, 2009

[14] GROVES, T., KNOCKEL, J., SCHULTE E, *BFS vs. CFS - Scheduler comparison* 11/12/2009

[15] MCKENNEY, P., *Deterministic synchronization in multicore systems: the role of RCU* 2009

[16] EHRINGER, D. *Tha Dalvik virtual machine architecture* Marzo 2010

[17] MONGIA, B. S., MADISETTI, V. K., *Reliable real-time applications on Android OS*

[18] YUNAN HE, CHEN YANG, XIAO-FENG LI., *Improve Google Android user experience with regional garbage collection*

[19] BRANDOLESE, C., BELLASI, P. *Evoluzione dei sistemi embedded:verso architetture multi-core* 21/06/2012 (visto il 15/12/2012)

[20] CESATI, M., Slide del corso *Sistemi embedded e real-time* http://sert13.sprg.uniroma2.it/calendario.html (viste il 15/12/2012)

[21] *Real-Time Maude Case Study: The CASH scheduling algorithm* http://heim.ifi.uio.no/peterol/RealTimeMaude/CASH (viste il 15/12/2012)

[22] LELLI, J., *SCHED_DEADLINE v7* https://lkml.org/lkml/2013/2/11/373 (visto il 15/02/2013)

[23] LEE, B., *CPU operation (kernel & user) & system calls* http://www.etpenguin.com/cgi-bin/index2file.cgi?/pub/OS /Linux/Linux/_CPU_Operation.txt (visto il 10/01/2013)

[24] KOLIVAS, C., *FAQS about BFS* http://ck.kolivas.org/patches/bfs/bfs-faq.txt, 05/09/2010 (visto il 20/01/2013)

[25] *PCLinuxOS Kernel* http://pclinuxoshelp.com/index.php/Kernel (visto il 20/01/2013)

[26] *Zenwalk 7.2 Live Edition*http://www.zenwalk.org/modules/news/ article.php?storyid=146 22/10/2012 (visto il 20/01/2013)

[27] *Linux Scheduler – CFS and Virtual Run Time (vruntime)* http://oakbytes.wordpress.com/2012/07/03/ linux-scheduler-cfs-and-virtual-run-time/ 03/07/2012 (visto il 15/01/2013)

[28] *Cpu schedulers compared* http://repo-ck.com/bench/ cpu_schedulers_compared.pdf 20/10/2012 (visto il 25/01/2013)

[29] KOLIVAS, C. *The Brain Fuck Scheduler* http://ck.kolivas.org/patches/bfs/3.0/3.7/3.7-sched-bfs-427.patch (visto il 25/01/2013)

[30] *RCU Linux Usage* http://www2.rdrop.com/users/paulmck/RCU/ linuxusage.html (visto 25/01/2013)

[31] CORBET, J., *The SLUB allocator* http://lwn.net/Articles/229984/ 11/04/2007 (visto 25/01/2013)

[32] *Sorgenti del kernel 3.7.6* /Documentation/robust-futexes.txt

[33] *Sorgenti del kernel 3.7.6* /include/linux/completion.h

[34] *Sorgenti del kernel 3.7.6* /mm/slab.c

[35] *Sorgenti del kernel 3.7.6* /mm/slob.c

[36] *Android* http://it.wikipedia.org/wiki/Android (visto il 05/02/2013)

[37] *Android software Stack & native application architecture* http://android-app-tutorial.blogspot.it/2012/08 /architecture-system-application-stack.html (visto il 07/02/2013)

[38] *Stack based vs register based virtual machine architecture, and the Dalvik VM* http://markfaction.wordpress.com/2012/07/15/ stack-based-vs-register-based-virtual-machine-architecture -and-the-dalvik-vm 15/07/2012 (visto il 05/02/2013)

[39] *Google I/O Dalvik VM internals* https://sites.google.com/site/io/dalvik-vm-internals/ 2008 (visto il 05/02/2013)

[40] *API Guides* http://developer.android.com/guide/components/index.html (visto il 05/02/2013)

[41] *Compatibility program overview* http://source.android.com/compatibility/ overview.html (visto il 05/02/2013)

[42] *Android 4.2 compatibility definition* http://static.googleusercontent.com/ external_content/untrusted_dlcp/source.android.com/it/ compatibility/4.2/android-4.2-cdd.pdf

[43] *Android API reference: AsyncTask* http://developer.android.com/reference/ android/os/AsyncTask.html (visto il 05/02/2013)

[44] *Android API reference: Process* http://developer.android.com/reference/ android/os/Process.html (visto il 05/02/2013)

[45] *Android API reference: PowerManager* http://developer.android.com/ reference/android/os/PowerManager.html (visto il 05/02/2013)

[46] GARGENTA, M.*Android internals* http://marakana.com/s/post/271/ andevcon_android_internals 08/04/2011 (visto il 10/02/2013)

[47] BRADY, P., *Anatomy & physiology of an Android* https://sites.google.com/site/io/anatomy–physiology-of-an-android (visto il 10/02/2013)

[48] *Android training: performance tips* http://developer.android.com/training/ articles/perf-tips.html (visto il 10/02/2013)

[49] *Android API reference: ActivityManager.RunningAppProcessInfo* http://developer.android.com/reference/android/app/ActivityManager. RunningAppProcessInfo.html#importanceReasonCode (visto il 10/02/2013)

[50] *How to configure Android's *internal* taskkiller* http://forum.xda-developers.com/showthread.php?t=622666 (visto il 10/02/2013)

[51] CHENG, B., BUZBEE, B. *A JIT Compiler for Android's Dalvik VM* http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html (visto il 15/02/2013)

[52] *Android API guides: Renderscript computation* http://developer.android.com/guide/topics/renderscript/compute.html (visto il 15/02/2013)

[53] SAMS, R. J., *Evolution of Renderscript performance* http://android-developers.blogspot.it/2013/01/evolution-of-renderscript-performance.html (visto il 15/02/2013)

[54] DUBROY, P., *Memory management for Android apps* http://www.google.com/events/io/2011/sessions/memory-management-for-android-apps.html (visto il 15/02/2013)

[55] *Android e Windows Phone: piattaforme dominanti nel 2015?* http://www.androidworld.it/2011/03/31/android-e-windows-phone-piattaforme-dominanti-nel-2015-41842/ 31/03/2011 (visto il 25/02/2013)

[56] *Ubuntu Platform overview* http://developer.ubuntu.com/resources/platform/documentation/platform-diagram/ (visto il 25/02/2013)

[57] *Built for the phone industry* http://www.ubuntu.com/devices/phone/operators-and-oems (visto il 28/02/2013)

[58] PHIPPS, S., *Ubuntu Phone: not a moment too soon* http://www.infoworld.com/t/open-source-software/ubuntu-phone-not-moment-too-soon-210074-0 04/01/2013 (visto il 01/03/2013)

[59] *App design guides* http://design.ubuntu.com/apps/ (visto il 01/03/2013)

[60] *Ubuntu Phone OS al CES 2013* http://www.ubuntu-it.org/news/2013/01/13/ubuntu-phone-os-al-ces-2013 (visto il 01/03/2013)

[61] *Smartphone Ubuntu prima del 2014? No, la colpa è dei carriers* http://www.chimerarevo.com/smartphone-ubuntu-2014/ (visto il 01/03/2013)

[62] *HTML5* http://it.wikipedia.org/wiki/HTML5

[63] *Ubuntu Touch su Nexus 4,7 e 10 a confronto* http://www.lffl.org/2013/02/video-ubuntu-touch̃su-nexus-47-e-10.html#more (visto il 01/03/2013)

[64] *Devices* https://wiki.ubuntu.com/Touch/Devices (visto il 01/03/2013)

[65] *Ubuntu Phone OS: Canonical scende in campo nella telefonia* http://leganerd.com/2013/01/02/ũbuntuphone/ (visto il 01/03/2013)

[66] *Tutorial librerie Qt parte 1* www.sereno-labs.com/uploads/Qt-Tutorial_1.pdf